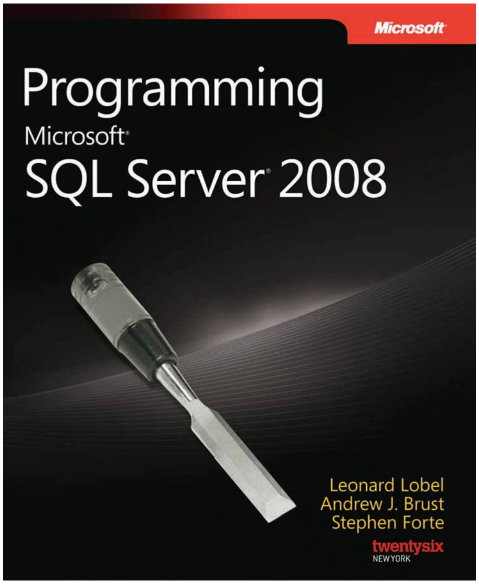


Programming Microsoft® SQL Server® 2008

Leonard Lobel,
Andrew J. Brust,
Stephen Forte
(twentysix new york)

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/12753.aspx

9780735625990

Table of Contents

Part I Core Fundamentals

3 Exploring SQL CLR 111

4 Server Management....161

5 Security in SQL Server 2008 189

Part II Beyond Relational

Part III Reach Technologies

Part IV Business Intelligence

16 Advanced OLAP 639

Chapter 14

Data Warehousing

—Mark Frawley

This chapter is all about data warehousing. If you've been avoiding this topic—dismissing it perhaps as being too advanced, esoteric, or abstract to be applicable—this chapter will help you cast those excuses aside and embrace data warehousing. The practical advice and guidance we give will empower you and your end users to glean more useful information and intelligence from your data. We will begin with an explanation of exactly what data warehousing is and why you should care about it, and then we'll show how to take advantage of specific Microsoft SQL Server 2008 data warehousing features.

Data Warehousing Defined

You're in good company if you wonder exactly what is meant by *data warehousing*—and indeed you might even wonder whether it has any *precise* meaning at all. The term has existed for almost two decades, and you might have seen a variety of definitions. Here is ours:

Data warehousing is both a vision of and a methodological approach toward organizing and managing enterprise data for the purpose of providing a trustworthy, consistent, integrated, and comprehensive data foundation for an enterprise's data-driven requirements and applications, both tactical and strategic.

Why does our definition not include any technical references? Well, that's just the point! While technology *is* essential to actually realizing the vision, data warehousing is not—or should not be—fundamentally about technology. It is about laying the data foundation needed to run an enterprise. *Run* as in making informed decisions. And *enterprise* rather than *business* because data warehousing is equally relevant whether the work is for-profit, not-for-profit, or in the public sector (a subtle distinction resulting from the unfortunate fact that the word *business* is embedded in the term *business intelligence*, or BI)—and, increasingly, whether the entity is small, medium, or large. Compared with what was true in the past, Microsoft's data warehousing–related offerings under the SQL Server product umbrella have made it particularly feasible for data warehousing goals to be attainable by small and medium-size enterprises. Of course, Microsoft continues to deliver industrial-strength data warehousing performance for the largest enterprises—especially with the 2008 release of SQL Server.

The Importance of Data Warehousing

Today, data warehousing in some form has become a *given,* a *must,* for running an enterprise of any significant size. At its best, it enables actual competitive advantage, but even when focused more tactically or departmentally, it is now considered essential to being competitive—as basic and essential as the general ledger or payroll system. While it is often difficult to quantify the benefits of data warehousing in terms of return on investment (ROI), no one these days seriously questions its value and necessity. As a database developer, you are likely to be involved with data warehousing in one way or another—if not directly, at least in interfacing to a data warehouse. So it's important for you to understand what data warehousing is all about.

Developing a data warehouse is in some ways a very different undertaking from traditional online transactional processing (OLTP) database development, with which you are probably more familiar. Two of the most notable differences are that data warehousing essentially emphasizes *data* and its relationships—as opposed to the emphasis on *process* found in the typical OLTP application—and that hard experience by practitioners has evolved specialized ways of modeling data that are particularly useful in achieving the goals of data warehousing.

Even if your role is primarily technical, you will be able to do a much better job of building or interfacing to a data warehouse if you know something about these differences from OLTP and the reasons for them. This will also help you appreciate the perspective of decision makers who rely on accurate data storage and analysis (see the next chapter), which will be very likely different from that of typical OLTP application stakeholders.

Data warehousing is an essential foundation for what has come to be known as business intelligence (BI). We'll learn more about the close relationship between data warehousing and BI later in this chapter, but for now, appreciate that they are *not* synonymous. At the same time, in keeping with our earlier observation, mentally substitute *enterprise* when you hear *business.*

The remainder of this chapter consists of five sections that build upon one another as we progress through our treatment of data warehousing. Instead of immediately focusing on technical details and step-by-step procedures in SQL Server 2008, we review the history leading up to why data warehousing is today a distinct practice and how SQL Server 2008 represents an excellent data warehousing platform.

The first section, "What Preceded Data Warehousing," focuses on the origins of data warehousing to help you appreciate why data warehousing emerged as a distinct practice responding to industry issues. The second section, "Data Warehouse Design," describes the two principal approaches to data warehouse design. The third section, "What Data Warehousing

Is Not," considers various terms often confused with data warehousing and gives them distinct definitions. The fourth section, "Practical Advice About Data Warehousing," alerts you to various common but nonobvious issues that you might encounter when building a data warehouse. Last, the fifth section, "SQL Server 2008 and Data Warehousing," discusses SQL Server 2008–specific details as they relate to data warehousing.

With this ambitious agenda to cover in just a single chapter, we will not actually tell you much about "how" to build the perfect data warehouse—dozens of entire books are available for that. Rather, what we aim to provide is a unique combination of background, clarification of terms, identification of tricky spots, and finally some technical details about the specific data warehousing platform offered by SQL Server 2008.

Data vs. *Information*

At the risk of sounding pedantic, fully appreciating why data warehousing is valuable requires drawing the distinction between *data* and *information*. Data consists of recorded, characterized "facts"—for example, sale amounts initiated by customer A at store B on date C, paid for with credit card D. These facts are the amounts of the sale (numbers), while the characteristics give these numbers meaning or context. This is the sort of transactional data typically captured by an operational application.

Such characterized facts are essential, but *information* involves interpreting facts, identifying the relationships between them, and finding the more abstract "meaning" (if it exists) implied by them. Each characteristic, such as customer, store, date, and so on, could serve as a predicate in a query. For example, what is the pattern of sales vs. store for this customer? Or what stores have the highest sales by date? Of course, there are countless others. These sorts of questions are *higher order*, or *value adding*, because their answers enable informed decision making for the future, as opposed to mere question answering of the sort that a customer service representative might do from the facts themselves (for example, when answering the question, "what is this charge on my statement that I don't recognize?").

This might not seem an important distinction, but historically, it often simply wasn't technically feasible to assemble the available data in a form suitable for informed decision making. Often, what passed for that instead was instinct and educated guesswork. In contrast, data warehousing emphasizes organizing, standardizing, and formatting facts in such a way as to enable deriving such "information" from them. Building on that, BI is then concerned with defining, extracting, delivering, and acting on that information.

What Preceded Data Warehousing

Depending on your experience, you might remember the term *electronic data processing*, also known as EDP or DP, which was used to describe the use of computers in enterprise applications for much of the 55+ years of computing history. Over the last 15 to 20 years, the term has morphed into today's *information technology*, commonly referred to simply as IT. Although unintentional, the timing of the change and the implication of the two terms could also stand for "pre–data warehousing" and "post–data warehousing."

Until the early to mid-1990s (when the client/server architectural paradigm reached its peak), the application of computers to enterprise needs had a strong emphasis on streamlining or automating manual clerical processes and relatively simple, repetitive high-volume tasks such as billing, payroll, inventory, and maintaining the general ledger (GL). Such applications were obvious initial targets for the application of computers in the business environment for at least three reasons:

- Their repetitive, highly constrained nature (making them relatively easy to model and suitable for automation)
- The presumed cost savings associated with that automation
- The technical feasibility given the state of the art at the time

Early input and output formats were very crude. For a long time, batch-mode processing—based on input via punched cards and output on green-bar lined printer paper—was the norm. Eventually, the state of the art advanced to allow interactive activities (giving us the now quaint and superfluous but persistent adjective *online*). Still, the application of computers to the enterprise remained largely driven by the aforementioned factors. A natural consequence was that each DP-targeted application was closely aligned with the operational process it supported, and marginally if at all with other processes. DP was about recording the basic facts of enterprise transactions while ensuring data integrity and then summarizing the results in fixed reports. The well-known term *online transaction processing* (OLTP) developed as a label for all of this.

Electronic data processing was an apt description of what computers and their users were doing during the pre–data warehousing period—processing data as transactions electronically (as opposed to manually)—and also what they were frequently *not* doing—turning data into information (as previously defined).

While this focus in many cases addressed operational needs adequately, it also led to a host of issues that impeded extracting a higher level of value from the data being collected. Data warehousing evolved, among other things, as a way of addressing these impediments. Let's explore how.

Lack of Integration Across the Enterprise

The emphasis on operational processes inevitably created nonintegrated, stand-alone applications. From both enterprise and technical perspectives, each application defined essential entities as it saw fit—not just the entities unique to itself but also those "master data" entities such as customers and products that exist across the enterprise. There was typically no common understanding of what was meant by these key entities, so each application kept its own version, leading to lots of data duplication.

With this state of affairs, it was difficult or impossible to create a meaningful enterprise-wide view of just about anything. When attempted, such views were necessarily at a high level of summarization, time-consuming, and expensive to create and therefore were created only infrequently. Enterprise decision making, especially at the operational and tactical level, still depended greatly on intuition, experience, and instinct. It often simply wasn't possible to base decisions on hard, accurate, up-to-date information. Late in the pre–data warehousing age, there were attempts to address this in the form of applications known as executive information systems (EIS) and decision support systems (DSS). These were generally ineffective because relative to their cost, they didn't deliver enough value to their small, high-level audience.

Management Reporting and the GL

The one application that typically *was* enterprise-wide was the general ledger (GL). Every other major application concerned with financial information (which was many, if not most applications) had to feed accounting entries to the GL. As a result, the GL often was the single point of integration between applications because it existed and had those connections already. Also, it was accepted as an enterprise-wide single version of "the truth" by its very nature. For these reasons, most early attempts at enterprise-wide reporting were driven from the GL.

There was value in this, but there were grave limitations as well. A GL is not well suited to "management reporting," except possibly at the highest aggregated levels, such as annual report line items. Management reporting is mostly focused on measurements of enterprise performance at much lower levels, levels which are irrelevant to the concerns of a GL—such as the profitability of specific customers. Yet once the GL became the single point of integration and thereby the source of management reporting, it started getting abused. All sorts of accounts and subledgers to support detailed management reporting proliferated in the GL, and modifications to the GL interface of source systems were made to feed them. Over time, this situation had a tendency to collapse under its own maintenance weight, especially when the GL chart of accounts needed to be restructured in the event of a merger. One of the impetuses of data warehousing was to address all this by providing a separate, appropriate environment for management reporting.

Little or No Standardized Reference Data

Closely related to lack of integration, there typically existed no single, agreed-upon "system of record" for key or master referential data across the enterprise, such as customer and product. Problems that stemmed from this included incomplete and inaccurate data, duplicated data entry (and resultant errors), and wasted effort synchronizing multiple versions from different applications. Most important of all was the inability to derive, except possibly at great effort, a consistent, comprehensive, and up-to-date view of the enterprise. In addition to these obvious consequences were some less obvious ones—for example, the embarrassment of severing a relationship with a customer who is unprofitable in one region but is overall very profitable, because you could not see the "big picture" of all your relationships with the customer across all regions, products, and organizational units.

To be sure, these problems and the reasons behind them were well recognized by the DP department and by the operational level of the enterprise almost from the beginning, and this led to attempts to create "master file" versions of the most important referentials—typically, customers, rates, products, and the organizational hierarchy. But technical limitations, political turf battles, and a lack of recognition at senior management levels of the costs of this fragmentation generally kept such efforts suboptimal.

Lack of History

Operational applications (let's call them "OpApps") by their very nature tend to neither require nor maintain historical data going back very far—often not more than a year or two. There are exceptions of course, such as an application that manages mortgage loans at a bank or life insurance at an insurer. These are certainly operational in nature and must also retain historical activity going back even decades perhaps. But in most cases, OpApps maintain a minimum of history in order to optimize their OLTP performance and minimize storage cost, and because there is simply no requirement to do more.

In any case, within the same enterprise, OpApps differ in the length of history maintained, its periodicity (that is, hourly, daily, weekly, monthly, and so on), and the way changes in referential data over time are handled (that is, whether a history of changes is maintained, and if so, on which attributes, and how many versions; for example, is the history of marital status or address of a customer maintained). These differences make integrating the historical data of multiple OpApps difficult, to say the least.

Data Not Optimized for Analysis

There are more significant differences between OpApps and analytical applications ("AApps," for short). As described so far, OpApps—especially in the pre–data warehousing era—were and still are concerned mainly with reliably recording the facts of current transactions. They have limited concern with past history or with other OpApps, which is why they came to be referred to as "islands of automation."

In contrast, AApps are concerned with "digesting" OpApp data to provide actionable insights, predictions, and an apples-to-apples view of the entire enterprise. Sometimes such applications even combine internal and external data, such as benchmarks regarding competitors, providing a view of how the enterprise looks in a larger context. Achieving these goals requires solving all kinds of problems that OpApps do not need to be concerned with. In addition to these general differences, here are some more specific ones:

- Given their uses, OpApps are physically optimized for insert, update, and delete operations, while AApps require read or query optimization.
- The amount of data required to answer a typical OpApps query is quite small, while the amount required to answer a typical AApp query can be huge. Imagine the amount of atomic data that must be digested to answer a query such as "Who were the top 5 customers by purchases for 2007, and what were the top 5 products purchased by each of them?"
- Among the various OpApps that must be integrated for an enterprise-wide view, there are many impediments to integration, in addition to those mentioned earlier. Here are a few:
    - Entities that mean the same thing but that are named differently
    - Entities that mean different things but that are named the same
    - Different encodings of the same thing (for example, country codes)
    - Different scale and precision of measures
    - Different lengths of descriptive text for the same thing
    - Different conventions for the primary key of the same entity
    - "Smart keys"—where information is encoded in primary keys

As a Result…

- Creating any particular view of enterprise data, especially one integrated across multiple applications, was a very technical undertaking that only the DP staff could perform. Usually, there was a large backlog of requests for such views or reports.
- Many such requests (the fulfillment of which might have helped run the enterprise better) never materialized in the first place. That was because users knew that by the time the DP department could fulfill them, it would be too late to meet the business opportunity.

- Each request that was fulfilled was usually implemented through a new report or extract, even if its requirements varied only slightly from an existing one. Given the technology of the time, even something as simple (as we would consider it today) as aggregating the data at a different level—say, quarterly rather than monthly—resulted in a new report. Further, even when a report already existed that could fulfill a request, there was typically no way to know that because no effective metadata was maintained about existing reports—and so a new one would be created.
- Every report or extract would become permanently enshrined in the system infrastructure, forever. There was often no way to track who was using what report for what purpose (if it was being used at all), so once a report was running, it was easier and safer to just keep supporting it.
- Eventually, there were extracts of extracts—one "report" would become the source for another. Keeping track of the dependencies became difficult if not impossible.

It should be obvious how all this represented a huge maintenance nightmare. But up through the early 1990s, this situation was all too common in the average "DP shop," and it just kept getting worse. It became increasingly evident that this was a crisis in the making, and what we today call data warehousing was born in response.

In fairness, it should be noted that there *were* efforts to build what effectively were data warehouses long before the term was coined. But in those days, such efforts essentially re-invented the wheel each time. They could not benefit from what is available today now that techniques have matured and become codified and, thanks to the advent of the Internet, shared. It is also true that hardware advances in the form of drastically lower storage costs and fantastically improved CPU capacities have had a profound impact on the practice of data warehousing and are essential to its viability today.

Data Warehouse Design

The preceding discussion gives you an idea of the issues that data warehousing evolved to address. In this section, we only scratch the surface of design considerations in bringing a data warehouse into existence and hope that will whet your appetite to learn more. Fortunately, it has never been easier to learn more about data warehousing than it is today.

Note The value of data warehousing was not always widely accepted. In its early days, it was viewed suspiciously and considered to be just a fad or an expensive waste of time by many IT practitioners. At best it was thought of as "nice to have" and something that only the largest, best funded, and mostly for-profit enterprises could consider. Fortunately, none of this is true any longer.

Building a data warehouse requires addressing a myriad of technical *and* nontechnical issues, including the following:

- Determination of enterprise goals and objectives to be served by the data warehouse and gaining organizational buy-in for them.
- Identification of the various audiences for the data and their varying requirements.
- Addressing of latency requirements with the appropriate data architecture.
- Extract, transform, and load (ETL)—the process and tools by which data is extracted from source OpApps, cleaned and otherwise transformed as needed, and then loaded into the data warehouse. SQL Server Integration Services (SSIS) is Microsoft's primary ETL tool for data warehousing.
- Design of entitlement, backup, mobility, scalability, delivery, and training schemes.
- Methods of end-user access to the information, including the distinction often made between reporting and analysis. The tools and products for this usually receive a disproportionate amount of attention in a data warehousing project because they are so visible.
- The embedding of an organizational ethos that the data warehouse will constantly evolve with the ever-changing needs it supports. The effort is never "done."

The primary goal of any data warehouse is to integrate data from disparate sources into a centralized store (at least logically speaking), in a form that can be used across the enterprise for decision support by all who need it. Merely dumping all the data from various standalone applications into a common database is not the sort of integration we mean. Rather, a data warehouse requires a schema of some sort to which all the data brought in is made to conform. The data also needs to be "clean"—meaning that all the different ways of representing the "same" thing in the various source systems have been converted to a single consistent form. Both of these tasks are ETL responsibilities, as previously mentioned.

Based on what we've said so far, the 35,000-foot view of a data warehouse is shown in Figure 14-1.

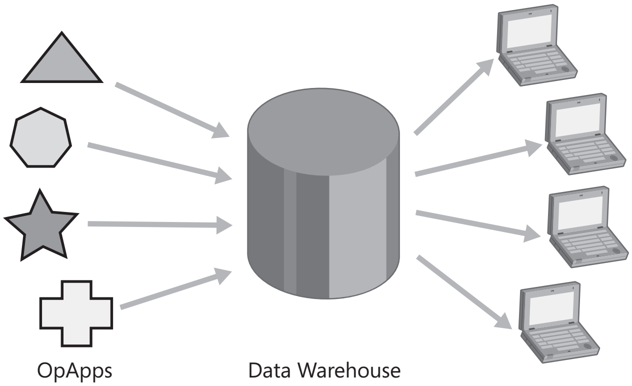


FIGURE 14-1 The generic data warehouse architecture

With this background in place, we can now consider the two predominant data warehousing architectures guiding practice today.

The Top-Down Approach of Inmon

William Inmon is recognized as "the father of data warehousing," having invented the term in 1990. The data warehousing features he characterized can seem self-evident today, but no one had codified them previously as he did. According to his definition, the essential characteristics of data in a data warehouse are as follows:

- **Subject-oriented** Major entities are common across multiple OpApps. Customer, Product, Shipment, and Account are typical subject areas.
- **Integrated** Data sources are consistent with one another along common themes.
- **Nonvolatile** Data, once loaded, is usually never changed (updated or deleted).
- **Time-variant** Time is part of the key to everything—"as it was at this point in time," also known as "history," is preserved.

These features enable the previously stated goals of any data warehouse.

While an oversimplification, the Inmon style of data warehousing presumes that an enterprise data model has been or will be created—one that identifies all the "subject-oriented" entities common across multiple OpApps, the required numeric measures, the required detail level of each, and the relationships between them. It is posited that the logical data model representing this within the data warehouse is a normalized relational model of the sort associated with OLTP applications. Inmon refers to this as the "enterprise data warehouse" and to the data as being "architected." The emphasis is on a centralized, normalized data store.

Since the typical complexity of a normalized model does not lend itself to direct query from ease of use and performance perspectives, this architecture also posits various *datamarts*, which are additional derived databases whose structure is optimized for query, and which generally contain only aggregated data derived from the data warehouse. The key point is that their architecture is secondary and separate from the data warehouse proper. A refinement of Figure 14-1 that represents Inmon's datamart concept is shown Figure 14-2.

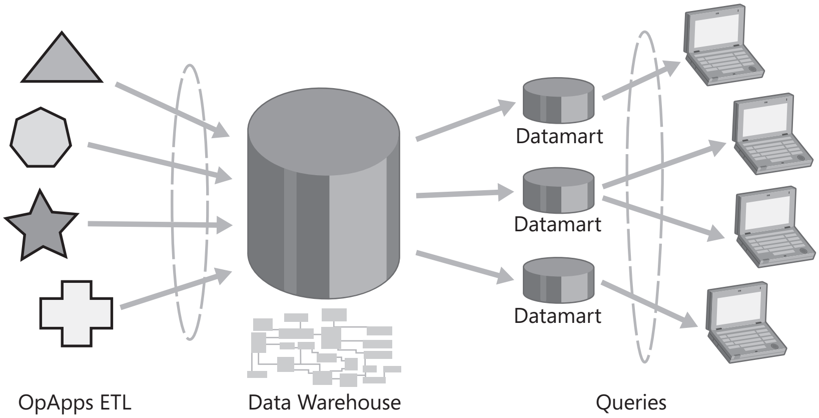


FIGURE 14-2 An Inmon-inspired data warehouse

Because this approach generally insists that a large-scale model already exists or will be created before construction of the data warehouse begins, it is usually characterized as *top-down*.

Inmon has written several books elaborating the principles and refinements of this architecture, and along with Claudia Imhoff (a long-term associate), he has elucidated an even larger architecture, the Corporate Information Factory (CIF), of which data warehousing is only a part. Space constraints preclude us from delving into further detail about the Inmon and CIF approaches. We do want to make two points before moving on, however.

The first you are probably already thinking—that requiring the existence or creation of an enterprise data model is impractical in many organizations. It *has* been successfully done, typically in larger enterprises, but many would find it impossible to justify the time and expense required to develop the model (with nothing to show at the end but documentation). No doubt when it *can* be done, it lays a very powerful foundation for informational applications, but in many cases, it is not feasible.

The second point is that many find this approach relatively abstract—useful in articulating high-level architecture but less helpful with practical details during actual development. The next approach to data warehousing that we'll discuss, at the other end of the design spectrum, evolved to address both these realities.

The Bottom-Up Approach of Kimball

From the mid 1990s to the present, Ralph Kimball has publicized an alternative to the Inmon approach to data warehousing, the heart of which he called the *Dimensional Model*. If the Inmon approach can be called top-down, Kimball's is definitely bottom-up, although both advocate a step-by-step approach. Just as Inmon articulated and formalized concepts that were already in use by practitioners, Kimball codified several practices already in use but lacking an integrative vision.

The first is the Dimensional Model, held to represent the most elegant tradeoffs between end-user intelligibility, ease of use, good performance for both predefined and ad hoc queries, and easy extensibility. The second is the idea of building the data warehouse incrementally, something most enterprises find much more palatable than the all-at-once, "big bang" approach implied by Inmon's architecture. A key part of this is the concept of "conformed dimensions" (which we'll define in a moment) to ensure that each new incremental data warehouse development could be integrated with what was already built, as opposed to each effort becoming the next-generation "island of automation," or as it is usually called today, "stovepipe," application. Third, Kimball emphasizes implementation practicality, with very specific advice on a host of data design issues advanced through his books, Web site, regular seminars, and training offerings.

Many indeed seem to find this approach desirable, as evidenced by the fact that most data analysis tools on the market today, including Microsoft SQL Server Analysis Services (which we cover in Chapters 15 through 18), have a definite affinity for the Dimensional Model. For this reason, as well as because it is less abstract, we will devote the rest of this section to an overview of this approach.

Important Inmon and Kimball are by far the best-known data warehousing pundits. For better or worse, because their approaches are often seen as so different, each has developed a "camp" of supporters who criticize each others' views of data warehousing best practices with sometimes religious zeal. Nonetheless, both share an emphasis on adhering to an architecture for the data warehousing design and on a step-by-step approach to design and construction. Most data warehousing projects in fact combine elements of the two approaches, which is as it should be, because each has excellent ideas to contribute. This is why it is prudent for you to be aware of them both.

This section does not purport to teach the Kimball approach. Space permits us merely to expose you to a few key concepts associated with it. This should make your further investigations easier and more effective.

Terminology

You should be aware of several useful data warehousing terms that—while closely associated with (if not always originated by) Kimball and the Dimensional Model—have come to be more broadly understood due to their representation in many tools (especially OLAP tools). You'll see most of these terms again in the chapters that cover SQL Server Analysis Services (Chapters 15 through 18).

- **Measure** A typically numeric value of interest in reporting and analysis, such as price, balance, or inventory. As stored in a data warehouse, the relevant measures are defined by the industry of the enterprise and come from the OpApps that are its data sources. A measure is also characterized by *grain*, defined later in this list.
- **Dimension** The heart of the Dimensional Model, a dimension is variously described as an "axis of analysis" or a "what" qualifier. A dimension helps qualify a measure and give it context (discussed in the next section). In a query, a dimension can be part of the query result and/or part of the query constraints. The most fundamental dimension is *Time*, essential in almost any context. Others are industry-specific but typically include at a minimum *Customer*, *Product*, and *Geography*. Dimensions are typically recognized as referential or master data entities. A dimension is a collection of related values called *members*—for example, *2008* might be a member of the *Time* dimension and *John Smith* a member of the *Customer* dimension. In a Dimensional Model, the dimensions are considered to be independent of one another, even if they really are not. For example, *Customer* and *Product* are not independent, since not every customer buys every product, but by modeling each as a dimension, we treat them as if they are independent because doing so simplifies the conceptual model on which queries are based. Few if any dimensions have zero correlation with any other dimensions.
- **Hierarchy** A particular parent-child organization of members within a dimension. Each distinct set of parents is called a *level* of the hierarchy. For example, a *Time* dimension might have levels named *Year* and *Month*. The *Year* level might have members like *2007* and *2008*, while the *Month* level might have members like *Jan 2007* and *Jan 2008*, with parent members at the *Year* level of *2007* and *2008*. Hierarchies occur naturally in a wide range of applications and are nothing more than a way of grouping members for summarization. A hierarchy reflects the fact that different members of the same dimension represent different levels of detail.
- **Dimension table** A relational table containing (typically) one row per member of the dimension (depending on what form of history, if any, is maintained in the dimension). A dimension table usually has a minimum of two columns, one representing the key or identifier that uniquely defines members of the dimension and another giving a descriptive name for the member.
- **Fact table** A relational table that functions, from a data modeling perspective, as an associative entity between various dimensions. It contains one or more measure columns, and key columns of all related dimensions. It is populated (by ETL) in such a

way that the measure values are completely described by the related dimensional keys. A fact table is also characterized by its *grain* (defined later in this list), and all measures in the same fact table (should) have the same grain.

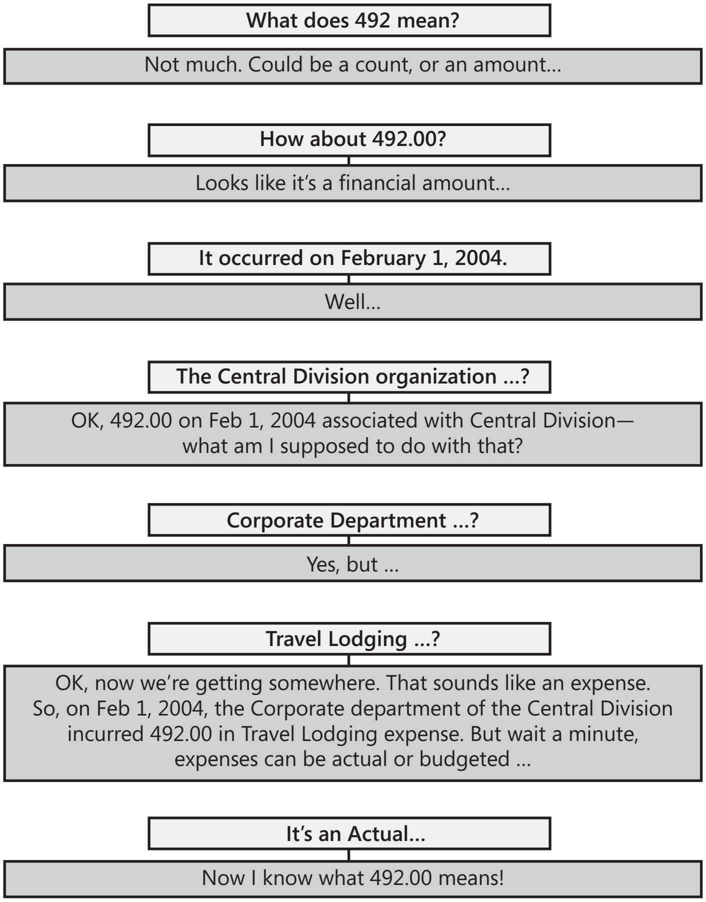
- **Star schema** Based on what an Entity Relationship (E/R) diagram of a fact table and its related dimension tables look like, this has become a generic term for that pattern (discussed later in this section).
- **Grain** A characteristic of a measure that is defined in terms of its related dimensions. Grain has two properties: first, precisely those dimensions that define the context of the measure; second, for each such dimension, the *level* within a hierarchy from the dimension that defines the level of detail of the measure. These two properties together define the measure's grain. For example, if all measures in a fact table pertain to values of the *Month* level of the *Year-Month* hierarchy of the *Time* dimension, the *Time* grain of that fact table is *Month*. The overall grain of the fact table, referred to as its *granularity*, is defined by such characteristics for all its dimensions.
- **Conformed dimension** A dimension, as previously defined, that has been designed and built in such a way that each star schema that includes the dimension can be meaningfully joined (logically) on such dimension. From a practical perspective, this means that all occurrences of such dimension in various fact tables mean the same thing—each includes exactly the same members, and each member has exactly the same meaning in relation to the facts whose context it helps define. Kimball refers to this state of affairs as the "Bus Architecture."

  It is not the case that each fact table using the dimension must use it at the same level (if it has a hierarchy). For example, if one fact table is at the *Year* level of the *Time* dimension and another is at the *Month* level, data from the two can still be meaningfully combined—it is simply necessary to aggregate the *Month* data to the level of *Year* first. Without conformed dimensions, various star schemas cannot be meaningfully combined along their common dimensions—in which case, the incremental approach to building up the data warehouse is not possible. Creating conformed dimensions is probably the most difficult part of the Dimensional Model approach, and where it most intersects with the Inmon approach—it is here that organizational agreement about which dimensions can be conformed, and what they will mean, must be secured. This is also where a lack of needed data (that is, at the required grain) in source OpApps will become apparent.

Note While the term *conformed dimension* concentrates on dimensions, the grain of the measures to be given context by such dimensions is equally important. To define conformed dimensions, there must exist measure definitions whose grain in the proposed conformed dimensions is the same in all existing or contemplated fact tables.

Context and the Star Schema

As mentioned earlier, dimensions provide the context of a measure. Figure 14-3 depicts an imaginary conversation that demonstrates how context is needed to make sense of data.



What does 492 mean?

Not much. Could be a count, or an amount…

How about 492.00?

Looks like it's a financial amount…

It occurred on February 1, 2004.

Well…

The Central Division organization …?

OK, 492.00 on Feb 1, 2004 associated with Central Division—
what am I supposed to do with that?

Corporate Department …?

Yes, but …

Travel Lodging …?

OK, now we're getting somewhere. That sounds like an expense.
So, on Feb 1, 2004, the Corporate department of the Central Division
incurred 492.00 in Travel Lodging expense. But wait a minute,
expenses can be actual or budgeted …

It's an Actual…

Now I know what 492.00 means!

FIGURE 14-3 Determining the context of a measure

Note Actually, do we really now know everything necessary to give 492.00 complete context? Not unless we make a further assumption. Can you guess what? Of course—what *currency* is this in?

Now let's diagram this conversation, as shown in Figure 14-4.

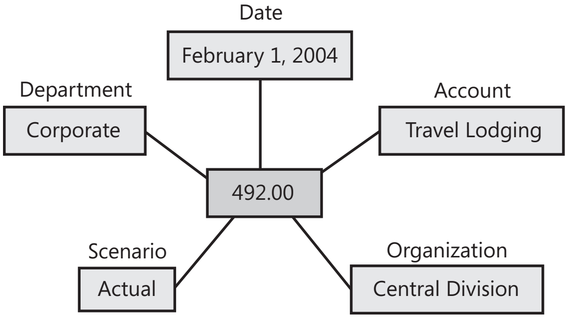


FIGURE 14-4 A representation of what we know about 492.00 (currency is assumed)

We can examine an actual implementation of the preceding example. Run the code shown in Listing 14-1 against the *AdventureWorksDW2008* sample database to retrieve our exact case.

LISTING 14-1 Querying *AdventureWorksDW2008* for the value of a particular measure

```
USE AdventureWorksDW2008
GO

SELECT
  dd.FullDateAlternateKey,
  do.OrganizationName,
  ddg.DepartmentGroupName,
  da.AccountDescription,
  ds.ScenarioName,
  ff.Amount
 FROM
  FactFinance ff
  INNER JOIN DimDate AS dd
   ON ff.DateKey = dd.DateKey
  INNER JOIN DimOrganization AS do
   ON ff.OrganizationKey = do.OrganizationKey
  INNER JOIN DimDepartmentGroup AS ddg
   ON ff.DepartmentGroupKey = ddg.DepartmentGroupKey
  INNER JOIN DimScenario AS ds
   ON ff.ScenarioKey = ds.ScenarioKey
  INNER JOIN DimAccount AS da
   ON ff.AccountKey = da.AccountKey
 WHERE
  dd.FullDateAlternateKey = '2/1/2004' AND
  do.OrganizationName = 'Central Division' AND
  ddg.DepartmentGroupName = 'Corporate' AND
  da.AccountDescription = 'Travel Lodging' AND
  ds.scenarioName = 'Actual'
```

Note The sample *AdventureWorksDW2008* database implements a schema that illustrates a Kimball-inspired data warehouse. Refer to this book's Introduction for instructions on locating and downloading this sample database.

From this query and the E/R diagram that represents the tables involved, we can see in Figure 14-5 what is meant by a star schema.

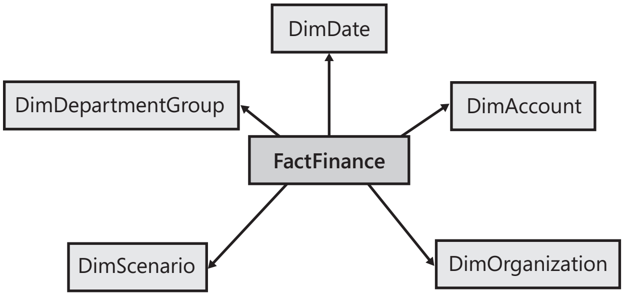


FIGURE 14-5 A star schema from *AdventureWorksDW2008*

Surrogate Keys

The *surrogate key* concept is not original to Kimball or the Dimensional Model, but it is something they strongly advocate. A surrogate key is a system-assigned, typically integer, primary key to a table. In SQL Server, the surrogate key would typically be an identity column, although sometimes a particular architecture might find it preferable to have a central *key generator* that gives out surrogate keys as needed. Surrogate keys have two important characteristics, as follows:

- They have no embedded encodings—that is, they are not "smart" keys. This makes them immune to changes in the source data that would plague nonsurrogate primary keys. One reasonable exception to this is the surrogate key of the *Time* dimension, where making the surrogate integer key smart by representing *YYYYMMDD* (when applicable to the grain of the fact tables) can make partitioning the fact tables much easier.
- As integers, they are the most efficient possible primary keys, both from performance and storage perspectives.

This concludes our brief review of the Kimball approach to data warehousing. You are strongly encouraged to consult the references at the end of this section, as well as appropriate Web searches, for a great deal more information. We'll close here with Figure 14-6, which illustrates what a data warehouse built to Kimball principles looks like. An important aspect to observe in this figure is that the data warehouse *is* the collection of star schemas—there are no separate datamarts, as in the Inmon approach. (And by the way, in an Inmon data warehouse, there is no objection to the datamarts following the Kimball architecture.) Although not shown in this figure, it is assumed that the various star schemas are not *disjoint*, meaning that wherever they share a functional dimension such as *Customer* or *Product*, they have been constructed in such a way as to actually share a single version of the dimension. When this is done, the data in the various star schemas can be validly combined along the common dimensions—a property derived from them having been constructed to be "conformable," in the parlance of the Dimensional Model.

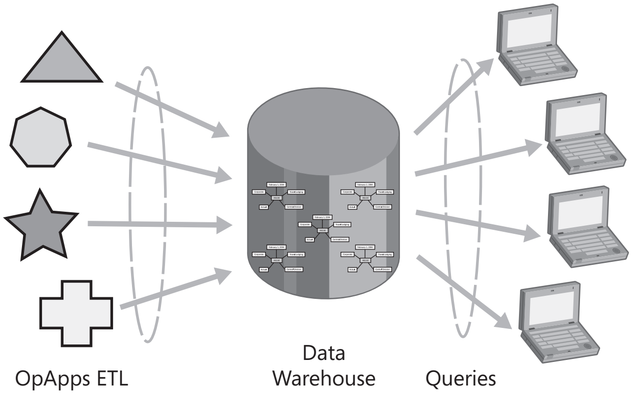


FIGURE 14-6 A Kimball-oriented data warehouse

What Data Warehousing Is Not

Much confusion exists in the literature and among practitioners because many terms are regularly conflated with data warehousing, even now when the maturity of the field should preclude this. A charitable view is that this was at least understandable in the past when the field was evolving rapidly in theory, practice, and product. But today, there ought to be more clarity, precision, and common understanding. In furtherance of this, we feel it is worth asserting that there are worthwhile distinctions still represented by certain overused and misused terms. This section provides a brief summary of some of these terms.

OLAP

The term *online analytical processing*, or OLAP, was coined by Dr. E. F. Codd (the originator of the relational model) in 1994 to distinguish a set of properties that analytical applications should satisfy (in contrast with his famous 1985 publication of "12 Rules" that a relational database management system should satisfy; see *http://en.wikipedia.org/wiki/Codd's_12_rules*). The term was intended to draw distinctions between the at-the-time well-known properties of OLTP applications and the less-well-defined properties of analytical applications. It is probably most valuable simply for emphasizing that such a distinction should be made. Today the term can be understood also as referring to a response to the limitations of spreadsheet-based approaches. While not strictly part of the definition, as a practical matter, cube-based technology is now usually associated with OLAP.

Note As with data warehousing, there were OLAP-like efforts long before the term OLAP was coined that were recognizable precursors, going back to the 1960s.

An OLAP application often, although not of necessity, draws its data from some form of star schema. The various OLAP tools on the market today form a spectrum in the degree to which they require a recognizable star schema as their data source. At one end, some tools can deliver OLAP functionality, with relatively simple calculations, from just about any data source with any organization, while at the other end are tools that can use only *cubes* (a data structure designed to facilitate fast analysis, further described in Chapter 15) as their data source (and hopefully can fully exploit their power). A data warehouse is very helpful as the source anywhere on this spectrum and is a virtual necessity on the cube-oriented end of it.

More Info The Fast Analysis of Shared Multidimensional Information (FASMI) test is a more precise, alternative definition of the properties that the term OLAP aspired to distinguish, developed by the authors of *The OLAP Report*. For a detailed definition of FASMI, as well as links to a wealth of other excellent OLAP information (much of it free), see *http://www.olapreport.com/fasmi.htm*.

In the context of SQL Server, Analysis Services is Microsoft's full-featured OLAP engine; it is covered in detail in Chapters 15 through 18.

Data Mining

The traditional way of extracting information from data requires a skilled analyst with a deep understanding of the enterprise who formulates ad hoc queries, the answers to which he or she think would be interesting—for example, "What was the impact of last month's sales promotion on sales?" or "Which stores in the top 10 by sales this year were also in the top 10 by sales last year?" In effect, the analyst forms hypotheses of cause and effect and then tests them against the data. To be effective, this rather hit-or-miss style of information discovery requires tools that permit easily formulating the queries and fast response so that the analyst can maintain his or her train of thought. OLAP technology is ideally suited for this.

In contrast, data mining is an approach in which correlations that might exist in a data set are automatically "discovered" using specialized data models and statistical algorithms. Because it is automated, it is more thorough in finding correlations, and it is unaffected by the prejudices and blind spots that an analyst would have using an ad hoc approach. The analyst still needs to evaluate each correlation found to determine whether it is meaningful or merely correlative, however.

In principle, data mining does not require a data warehouse for its source data. However, a well-crafted data warehouse with clean data could be an ideal source. The intended analysis and the allowable latency also affect whether a data warehouse as an analysis source is feasible. For example, in detecting credit card fraud, is the data warehouse updated often enough to be useful?

Starting with SQL Server 2000, Microsoft has invested much effort in giving SQL Server Analysis Services data mining capabilities that are much easier for relative nonspecialists to use than what has previously been available on the market. These capabilities are covered in detail in Chapter 18.

Business Intelligence

The term *business intelligence* (BI), coined by analyst Howard Dressner in 1989, has turned out to be quite popular. Today it is applied in so many contexts that you would be right to wonder whether it distinguishes anything anymore. Some argue that it doesn't, but we think that it still does. It *is* unfortunate that the *business* in BI obscures the fact that BI can be valuable in any enterprise, not just the for-profit ones implied by the *B*. So as suggested earlier, think *enterprise* intelligence when you hear *business* intelligence.

The most important thing to be clear about is that BI, properly understood, is not about any particular technology—although its implementation certainly depends on technology. BI is fundamentally a management approach and philosophy. Like most good ideas, its basic premise sounds so obvious when stated that it hardly seems worth noting: management decisions should be based on facts, not on educated guesswork, politics, or other subjective bases. Of course, management of an enterprise has always been based at some level on objective information—accounting being the most elemental form. But in the past, such objective measures, especially at the enterprise level, were at a summary level, produced infrequently (if periodically), rigidly structured, and incapable of easily revealing the detail from which they were derived.

BI aims to change all this by ensuring that information is accurate, reliable, updated as frequently as necessary, and readily accessible to whoever needs it, regardless of their level in the organization. One focus of BI *is* on the technologies required to achieve these goals, which generally include some form of data warehouse—hence the association. But the technology focus, especially on user interfaces (UIs), tends to receive disproportionate attention. An equally important focus should be on the vision of fact-based decision making that is supported by senior management and influences the way the enterprise will be run.

Initially, BI often faced significant resistance in the enterprise. If knowledge is power, losing control of knowledge feels like (and often is) losing power. BI threatened this with its emphasis on making information available to a much broader audience. Fortunately by now, the value of BI is recognized in most enterprises.

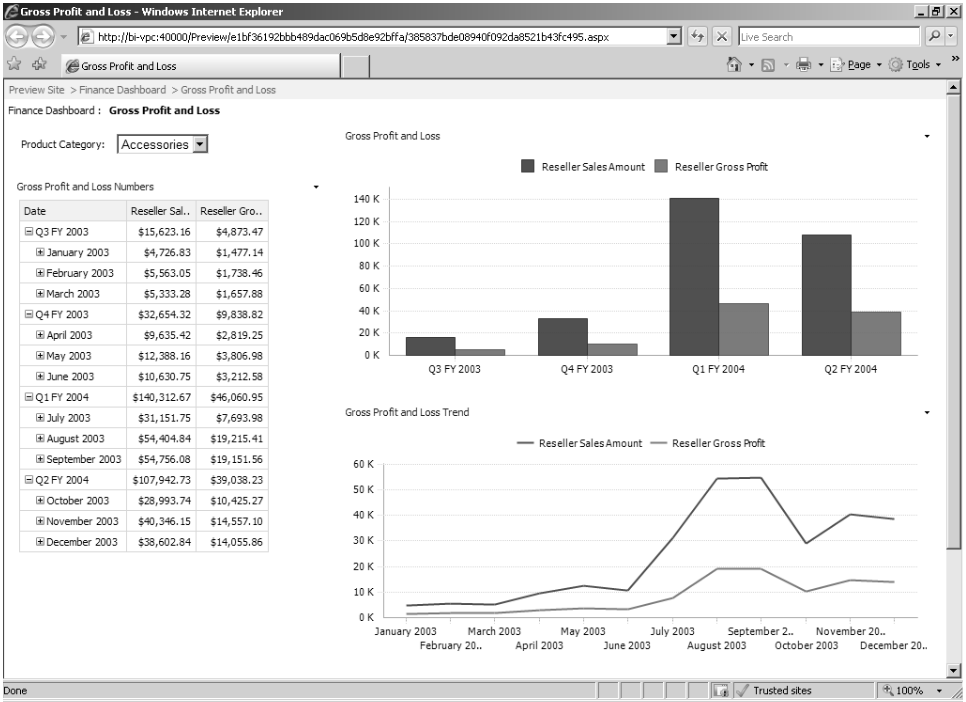
Last, we must mention that historically, many BI projects and their supporting data warehouse implementations have overpromised and underdelivered, giving BI a bad reputation for being expensive and risky. As a result, some are beginning to rethink the necessity of creating a data warehouse to support BI and instead are using existing reports and other existing data sources directly as BI sources. While this approach has its appeal, only time will tell whether it becomes an important theme in BI implementation.

Dashboards and Scorecards

The terms *dashboard* and *scorecard* are often used synonymously. They both represent information graphically, summarizing it with various elements showing relative magnitudes, trends, and other meaningful relationships. But they are not synonymous.

Dashboards

A dashboard, like its automobile namesake, displays measures without the context of related goals. It has a "just the facts" tactical orientation and is updated as often as necessary for the (typically) operational process that it supports. It is more generic than a proper scorecard in that it can display anything (including a scorecard). Figure 14-7 shows a typical dashboard.



Finance Dashboard : **Gross Profit and Loss**

Product Category: Accessories

Gross Profit and Loss Numbers

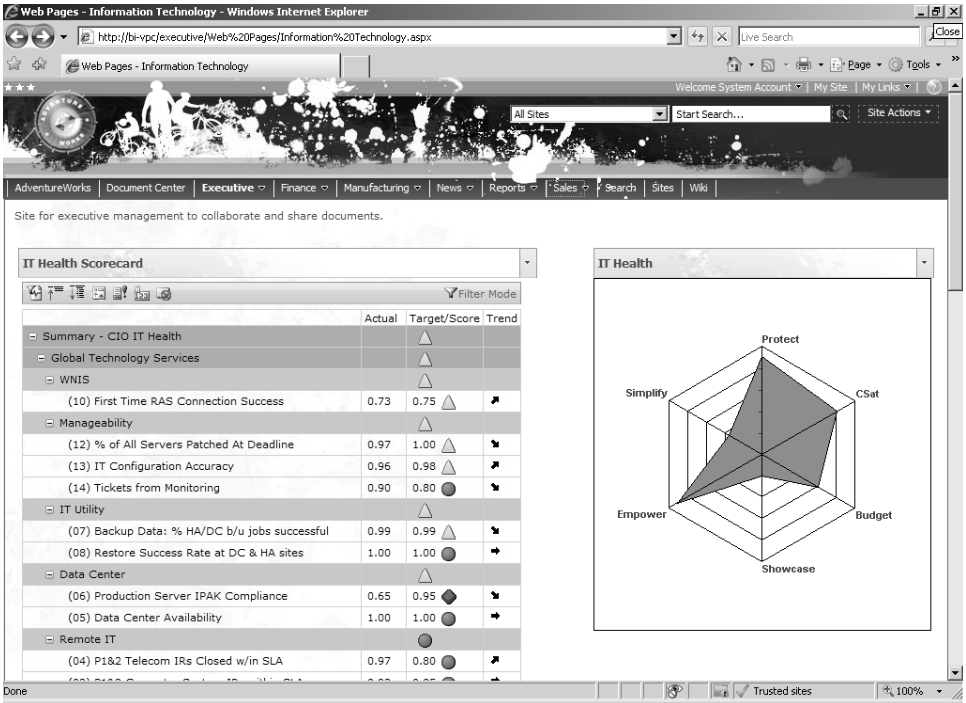
Date	Reseller Sal..	Reseller Gro..
⊟ Q3 FY 2003	$15,623.16	$4,873.47
⊞ January 2003	$4,726.83	$1,477.14
⊞ February 2003	$5,563.05	$1,738.46
⊞ March 2003	$5,333.28	$1,657.88
⊟ Q4 FY 2003	$32,654.32	$9,838.82
⊞ April 2003	$9,635.42	$2,819.25
⊞ May 2003	$12,388.16	$3,806.98
⊞ June 2003	$10,630.75	$3,212.58
⊟ Q1 FY 2004	$140,312.67	$46,060.95
⊞ July 2003	$31,151.75	$7,693.98
⊞ August 2003	$54,404.84	$19,215.41
⊞ September 2003	$54,756.08	$19,151.56
⊟ Q2 FY 2004	$107,942.73	$39,038.23
⊞ October 2003	$28,993.74	$10,425.27
⊞ November 2003	$40,346.15	$14,557.10
⊞ December 2003	$38,602.84	$14,055.86

FIGURE 14-7 A typical dashboard

Scorecards

A scorecard displays base measures in the context of related goals, objectives, or target measures and provides at-a-glance visual cues as to whether each such base measure is lagging, achieving, or surpassing its goal measure. Obviously, therefore, a scorecard is not possible unless such goal measures exist in addition to the base measures. A strategy must be devised for such goal measures to exist. It follows that a scorecard is strategic, whereas a dashboard is tactical and operational.

The term *key performance indicator* (KPI) is closely associated with scorecards. The traffic light and trend indicators in Figure 14-8 are KPIs. A KPI encapsulates a measure, a related goal measure, a calculation about the relationship of the two, and a graphic that expresses a "good or bad" indication based on the calculation.



Site for executive management to collaborate and share documents.

	Actual	Target/Score	Trend
Summary - CIO IT Health		△	
Global Technology Services		△	
WNIS		△	
(10) First Time RAS Connection Success	0.73	0.75 △	↗
Manageability		△	
(12) % of All Servers Patched At Deadline	0.97	1.00 △	↘
(13) IT Configuration Accuracy	0.96	0.98 △	↗
(14) Tickets from Monitoring	0.90	0.80 ●	↘
IT Utility		△	
(07) Backup Data: % HA/DC b/u jobs successful	0.99	0.99 △	↘
(08) Restore Success Rate at DC & HA sites	1.00	1.00 ●	→
Data Center		△	
(06) Production Server IPAK Compliance	0.65	0.95 ◆	↘
(05) Data Center Availability	1.00	1.00 ●	→
Remote IT		●	
(04) P1&2 Telecom IRs Closed w/in SLA	0.97	0.80 ●	↗

FIGURE 14-8 A typical scorecard

Goal measures are usually not defined at lower levels of detail. Consider the difference in grain between *Actual* and *Plan* measures—the former derive from individual transactions, while the latter are created at a much more summarized level, at least in the *Time* dimension. For this reason, scorecards tend to report at a more summarized level than dashboards, which is consistent with their strategic vs. tactical orientation. This in turn also means that changes occur more slowly, so scorecards are usually refreshed less often than dashboards. In a financial scorecard like the one shown in Figure 14-8, an *Actual* vs. *Plan* KPI exhibits all these principles and is seen as a traffic light in the *Plan* columns. Notice the *Trend* indicator, which is also a KPI that uses some calculation between prior-period *Actual* and *Plan* values.

Since SQL Server 2005, Analysis Services provides KPI objects that can be stored in cubes. They can be consumed and displayed by Microsoft Office Excel 2007, Microsoft Office SharePoint Server, and Microsoft Performance Point, each of which also allows creating and storing KPIs within its respective environment.

More Info See Chapter 16 for advanced OLAP coverage that includes KPIs.

Performance Management

Performance management is a relatively recent term that is a particular flavor of BI but rates its own discussion because of its currency in the literature and market today as a distinct entity. Performance management implies BI—but the converse is not true, because BI is the more general term. As noted earlier, BI's techniques can be focused in many different directions. Performance management is a specific application of BI. It is first about establishing organizational goals and objectives and ways of measuring progress toward meeting them—often using BI techniques to help determine what those goals and measures should be. Once these goals are established, it is then about gathering past, current, and projected performance, explicitly measuring these against the established goals, and widely disseminating how well goals are being met. This is usually achieved in the form of scorecards, which are again facilitated by BI tools and techniques.

The Balanced Scorecard (BSC) is a well-known example of performance management that predates the term. It is worth becoming familiar with the BSC approach, not least because it can help you better understand the factors driving enterprise strategy, and how to ensure that the strategy is enacted.

More Info Start by reading the seminal book that originated the term: *The Balanced Scorecard: Translating Strategy into Action*, by Robert S. Kaplan and David P. Norton (Harvard Business School Press, 1996).

Practical Advice About Data Warehousing

A data warehousing effort requires both theory and discovery. Although the theory associated with building a data warehouse could be considered a rather well understood topic today, practical experience still has much to offer. In this section, we'll look at a few of the data warehousing best practices that we have found most valuable.

Anticipating and Rewarding Operational Process Change

It is almost certain that a data warehousing effort will identify data elements and relationships essential to realizing the enterprise goals that are not currently captured in the operational processes. It is also likely that those who would be most directly affected in their day-to-day work by addressing this will feel that they have nothing to gain by doing so, and often something to lose. For example, an enterprise goal might be to capture which sales groups should get credit, and in what proportion, for working together to make a sale happen—the better to apportion the bonus pool of the sales force. Enabling this requires capturing information about which sales groups were involved at the time the sales transaction is recorded. This is information that is likely not currently available in the workflow of the back-office staff who record the transaction, and moreover, even if it is (or is made to be), the extra time it would take them to record it will reduce the number of transactions they can process per hour. They will most likely resist, given the impact on their productivity, unless this effort is officially recognized and proper incentives are put in place to motivate their cooperation.

Rewarding Giving Up Control

As suggested earlier in this chapter in the section "Business Intelligence," a successful data warehousing/BI effort often requires those who have traditionally been in control of key data to relinquish that control in the interest of the greater good. Any organizational change effort will threaten those who perceive themselves the losers in some way (often correctly), and it is only natural for them to resist the change. If the enterprise recognizes this and provides positive motivators to take this risk, the chances of success are increased. How feasible this is, of course, depends greatly on the organizational culture. The BSC approach can be particularly valuable in this regard.

A Prototype Might Not Work to Sell the Vision

Building a prototype or proof of concept (POC) for a data warehousing/BI approach is often recommended as a way to achieve buy-in from important stakeholders. It is easy to assume that a representative POC will do the trick. By *representative*, we mean that the important technical capabilities are demonstrated as feasible (such as whether particular relationships can be modeled successfully), even if this is illustrated with fictitious data such as the *AdventureWorksDW2008* database.

What you might not realize until it is too late is that stakeholders can find it difficult to appreciate such an approach, particularly when the POC is not based on measures they recognize or the values used are not realistic. If you hear people in your audience calling out "Hey, that number isn't right!" while you are demonstrating the POC, that's exactly what's happening. Logically, in a POC, it might not matter whether the data is accurate, but once your stakeholders lose interest or faith, it can be very difficult to regain. Focusing on such issues is also a favored tactic of those who oppose the data warehouse for whatever reason.

For a POC to have the best chance of success, it should be as realistic and as attuned to the work of the stakeholders who will be judging it as possible. This often runs counter to the idea that a POC requires a minimal investment, which is exactly why we are making this point. The data warehousing project can get shot down before it even gets off the ground with an ill-conceived POC.

Surrogate Key Issues

The value of using integer surrogate keys in a data warehouse was discussed earlier in this chapter in the section "Data Warehouse Design." But their use is not without issues, as described here:

- In general, surrogate keys should not be "smart"—that is, they should not have any significant meaning encoded in their values. However, an exception might be worth considering for the *Time* dimension. At the physical level, there can be value in the *Time* surrogate key taking the form *YYYYMMDD*, *YYYYMM*, or *YYYYWW* (where *Y*, *M*, *D*, and *W* are year, month, day, and week values), all of which are easily represented as an integer. Two reasons justify this violation of the normal best practice. First, if the *Time* surrogate key column is the first in the composite primary key of the fact table (as it usually should be) and the primary key has a clustered index, the fact data will be optimally organized for the *Time* constraint of the typical query—which is usually either a point in time or a range. Second, such a smart *Time* key will make it much easier to implement and maintain physical partitioning of the *Time* dimension at the relational database level.
- Surrogate keys can be generated in several ways, two principal ones being *IDENTITY* columns or a row-by-row assignment facility—for example, *SELECT MAX(Id) + 1*—using appropriate locking mechanisms. Regardless of the method, complications can arise in the typical multienvironment setting—that is, development, quality assurance (QA), and production. Assume that at the start of a development cycle, your development environment is refreshed from production. Then you also copy over ETL input files from production and run the ETL process in development (perhaps as part of a parallel test). Depending on how surrogate keys are assigned, there can be a good chance that the same data (from a business key perspective) is assigned different surrogate keys in development and production. This can greatly complicate reconciliation between the two.

Currency Conversion Issues

Particularly in larger, multinational enterprises, financial applications usually require currency conversion in order to compare similar items (apples to apples). Be aware that this is a subject fraught with business rule and design conundrums. Since SQL Server 2005, Analysis Services has provided features that can make implementation of currency conversion calculations in the cube easier.

But this does not address the issues we want to highlight here, which relate to the tension between designing for ad hoc, not-known-in-advance queries and *needing* to know something, possibly a lot, about likely queries, if a suitable design is to be derived. Issues around currency conversion illustrate this particularly well. There are no "right" answers to the following questions, but you would do well to consider all of them if currency conversion is in any way a part of your business perspective:

- What flexibility is required? Will there be one master currency in which all comparisons are expressed, several standard currencies, or in any existing currency?
- Closely related to the preceding questions, does it make sense to precalculate and store converted amounts, or must this be done on the fly?

  As with all rates and ratios, care must be taken where aggregation is involved to force the currency conversion to be at the appropriate leaf level of detail, followed by aggregation to the required summary level. The capabilities of your OLAP tool influence this greatly.
- Are converted amounts to be at the rate in effect at their original point in time only, or should amounts also be convertible based on the rates at any point in time?
- At what rates should future values (for example, *Budget*) be converted: the rates in effect when the budget is finalized, never after to be adjusted? Or should current rates be used, adjusting the projections every period? Must you be able to distinguish how much of a variance between *Actual* and *Budget* is due to currency conversion vs. changes in the *Budget* measure itself?

The design driven by answers to these business questions has profound effects on both the questions that can be answered later and the technical complexity required.

Events vs. Snapshots

There are two complementary approaches to data warehouse logical design: the event-driven approach and the snapshot approach. Both involve tradeoffs in complexity and in the sort of inquiries they can support.

On the one hand, it can be argued that everything of analytical interest in an enterprise can be represented as an *event*. Events are items like a payment or an order being received or a shipment getting delivered. Events by definition occur asynchronously at points in time. In principle at least, if all relevant events can be identified and captured, it is possible to deduce the state of affairs at any point in time, as well as how that state came to be. For some informational applications, this is critical. Constructing the point in time from events can, however, be exceedingly complex.

On the other hand, a snapshot-based approach does not record events at all. Instead, it simply periodically records the aggregate effect of events. Answering queries about the points in time where snapshots were taken is obviously much easier than it would be with a purely event-based approach, where the state at the point in time would need to be reconstructed.

These approaches sometimes need to be combined. For example, with an *Account* entity, often the only thing of interest is the account balance at periodic points in time, such as month-end. On the other hand, it is also imperative to be able to query each and every event (debit or credit) that affected the balance since the previous snapshot.

Events and snapshots have considerations in addition to which functional questions they support. There is the question of what the source system can provide in terms of either events or snapshots, which has an impact on how much work must be done in the data warehouse ETL to create one or the other. Also, a snapshot approach that takes a snapshot of everything, regardless of how much or little has changed since the last snapshot can lead to data proliferation and can be inefficient compared with an event-based approach when changes are relatively few—although this can be addressed with techniques such as Change Data Capture, detailed later in this chapter.

It is well worth spending considerable time during the design phase thinking through the implications of both approaches before determining the best choices for your requirements.

SQL Server 2008 and Data Warehousing

Earlier versions of SQL Server had new features related to data warehousing, most notably Analysis Services, Reporting Services, and in SQL Server 2005, certain features of SQL Server Integration Services such as the Slowly Changing Dimensions task. But these earlier versions had very little at the level of the relational engine specifically targeting the particular needs of data warehousing. SQL Server 2008 delivers new features that squarely target data warehousing, particularly in relation to making very large databases more manageable and cost effective. This section will review the most important of the data warehousing–oriented enhancements in SQL Server 2008, starting with the Transact-SQL (T-SQL) enhancements aimed at working with data warehousing.

T-SQL *MERGE* Statement

The *MERGE* statement is covered in more depth in Chapter 2 and is applicable to many more scenarios than data warehousing. We cover it here too because it is also very relevant to data warehousing, specifically in the ETL context.

The *MERGE* statement provides what's commonly referred to as *upsert*—meaning *update* the row if it already exists; otherwise, *insert* it. But there is more as well. *MERGE* requires a target table, which is joined in some relationship to a source table. The source table contains the data to be merged or synchronized with the target table. The *MERGE* statement supports up to three types of clauses defining the row-by-row action to be taken on the target table based on how it compares with the source table:

- ***WHEN MATCHED*** The row exists in both merge and target tables (performs an inner join and allows *UPDATE* or *DELETE*).
- ***WHEN NOT MATCHED BY TARGET*** The row exists in the source table but not the target table (performs a left outer join and allows *INSERT*).
- ***WHEN NOT MATCHED BY SOURCE*** The row exists in the target table but not the source table (performs a right outer join and allows *UPDATE* or *DELETE*).

More Info Each merge clause can also state constraints in addition to the implied join, such as another condition comparing column values between source and target. However, there are some very particular rules governing the use of multiple merge clauses and their various combinations. We cover those in the full treatment given to the new *MERGE* statement in Chapter 2.

In the data warehousing context, the *MERGE* statement is particularly suited to the maintenance of the dimension tables of star schemas. It is also very helpful in maintaining Type 1 slowly changing dimensions (SCDs), where changes simply overlay existing values, and Type 2 SCDs, where *MERGE* can do part of the job (a separate *INSERT* operation is still needed when an existing row is updated, to create the new version of it.) See the section entitled "Data Warehouse Design" earlier in this chapter for more details. (A full treatment of SCDs is beyond the scope of this chapter.)

In SQL Server 2008 Integration Services, *MERGE* can streamline and simplify the insert/update pattern that would be required under SQL Server 2005 Integration Services. Previously, the decision to insert or update in SQL Server 2005 Integration Services had to be based on a lookup of the source row using a Lookup task that was loaded with the target rows and two output data flows based on the failure or success of the lookup: one doing inserts and one doing updates against the target. With *MERGE*, the Lookup task is no longer needed, which simplifies the Integration Services package and avoids the performance, memory, and deadlock issues that can arise with the Lookup task if the target table is large.

Syntactically, *MERGE* requires two joinable tables or table-equivalents. (The target must be either a table or an updatable view; the source can be any table-equivalent.) For Integration Services, this means that the source table must exist or must be created in the package (as a temporary table, common table expression [CTE], or other equivalent).

The code in Listing 14-2 shows a series of representative T-SQL expressions using *MERGE* against the *AdventureWorksDW2008* database. Run each statement by hand as directed by the comments, followed by running the *MERGE* statement at the end. Note that *GeographyKey* is an identity column in *DimGeography*, so the column list must be explicit in the *INSERT* statement in the *MERGE* statement's *WHEN NOT MATCHED BY TARGET* clause. Also note that the ending semicolon is required to terminate the *MERGE* statement.

More Info All the data manipulation language (DML) statements in T-SQL (*INSERT*, *UPDATE*, *DELETE*, and *MERGE*) support an *OUTPUT* clause, which can be quite useful for archiving changed data. In addition, the new INSERT OVER DML feature in SQL Server 2008 enhances the *OUTPUT* clause with filtering capabilities. See Chapter 2 for details of the *OUTPUT* clause and INSERT OVER DML.

LISTING 14-2 Using *MERGE* for a data warehousing update

```
USE AdventureWorksDW2008
GO

-- Make a copy of the table.
SELECT * INTO DimGeographyTest FROM DimGeography

-- Create "Changes" table as another copy of same data.
SELECT * INTO Changes FROM DimGeography

-- If you now run the MERGE statement below, no changes will be reported. Note
-- the condition on the UPDATE clause, which prevents unnecessary updates.

-- Now force some UPDATES (53):

UPDATE Changes
 SET SalesTerritoryKey = 11
 WHERE SalesTerritoryKey = 10

-- Now running MERGE reports 53 updates.

-- Now force DELETES (empty table will effectively delete every row in
-- DimGeographyTest):

DELETE Changes

-- Now running MERGE will delete all 653 rows in DimGeographyTest.

-- Testing INSERT is left as an exercise for the reader.

-- MERGE statement:

MERGE DimGeographyTest AS dg
 USING (SELECT * FROM Changes) AS c
 ON dg.GeographyKey = c.GeographyKey
 WHEN MATCHED and dg.SalesTerritoryKey <> c.SalesTerritoryKey THEN
  UPDATE SET dg.SalesTerritoryKey = c.SalesTerritoryKey
 WHEN NOT MATCHED BY TARGET THEN
  INSERT (City, StateProvinceCode, StateProvinceName,
          CountryRegionCode, EnglishCountryRegionName,
          SpanishCountryRegionName, FrenchCountryRegionName,
          PostalCode, SalesTerritoryKey)
```

```
   VALUES (c.City, c.StateProvinceCode, c.StateProvinceName,
           c.CountryRegionCode, c.EnglishCountryRegionName,
           c.SpanishCountryRegionName, c.FrenchCountryRegionName,
           c.PostalCode, c.SalesTerritoryKey)
  WHEN NOT MATCHED BY SOURCE THEN
   DELETE
  OUTPUT $action, INSERTED.*, DELETED.*;
```

The deletion possibilities of *MERGE* would be rare in a data warehousing scenario except in single-instance fixes of erroneous data, but it is worth knowing about for that purpose alone. In general, beware of using *DELETE* with *MERGE*. If your source table is inadvertently empty (as it is eventually in our example), *MERGE* with a *WHEN NOT MATCHED BY SOURCE* clause specifying *DELETE* could unintentionally delete every row in the target (depending on what other conditions were in the *WHEN NOT MATCHED BY SOURCE* clause).

Change Data Capture

Like one use of *MERGE*, the new Change Data Capture (CDC) feature in SQL Server 2008 targets the ETL component of data warehousing. CDC is available only in the Enterprise edition of SQL Server 2008 (and of course, the functionally equivalent Developer and Evaluation editions).

Note SQL Server 2008 provides a number of change tracking features—each one tailored for a specific purpose. In particular, CDC addresses data warehousing, SQL Server Audit addresses security (see Chapter 5), and SQL Server Change Tracking targets synchronization of occasionally connected systems and mobile devices using ADO.NET Sync Services (see Chapter 13).

CDC is designed to efficiently capture and record relevant changes in the context of a data warehouse. Traditionally, detecting changes in an OpApp table that need to be applied to a data warehouse has required relatively brute force methods such as the following:

- For updates, using the *CHECKSUM* function as a shortcut to detecting inequality of columns between source and target rows (SQL Server only), or comparing time stamps.
- For inserts, outer-joining source and target rows and testing for *NULL* on the target.
- For inserts and updates, implementing triggers on the source table to detect changes and take appropriate action against the target, or performing a lookup (perhaps using an Integration Services Lookup task) to compare source against target and then driving the update or insert by the success or failure of the lookup.
- For inserts and updates, using the *OUTPUT* clause (SQL Server 2005 and 2008) or INSERT OVER DML (SQL Server 2008 only), which we cover in Chapter 2.

The CDC feature introduced in SQL Server 2008 provides a valuable new way of laying the groundwork for maintaining changing data in a data warehouse. Without resorting to triggers or other custom code, it allows capturing changes that occur to a table into a separate SQL Server Change Tracking table (the *change table*). This table can then be queried by an ETL process to incrementally update the data warehouse as appropriate. Querying the change table rather than the tracked table itself means that the ETL process does not affect the performance of applications that work with the transactional tables of your database in any way. CDC is driven by a SQL Server Agent job that recognizes changes by monitoring the SQL Server transaction log. This provides much better performance than using triggers, especially in bulk load scenarios typical in a data warehouse—and there's no code to write or maintain with CDC. The tradeoff is somewhat more latency, which in a data warehouse is often perfectly acceptable. Figure 14-9 depicts a high-level view of CDC architecture using an illustration taken from SQL Server Books Online.

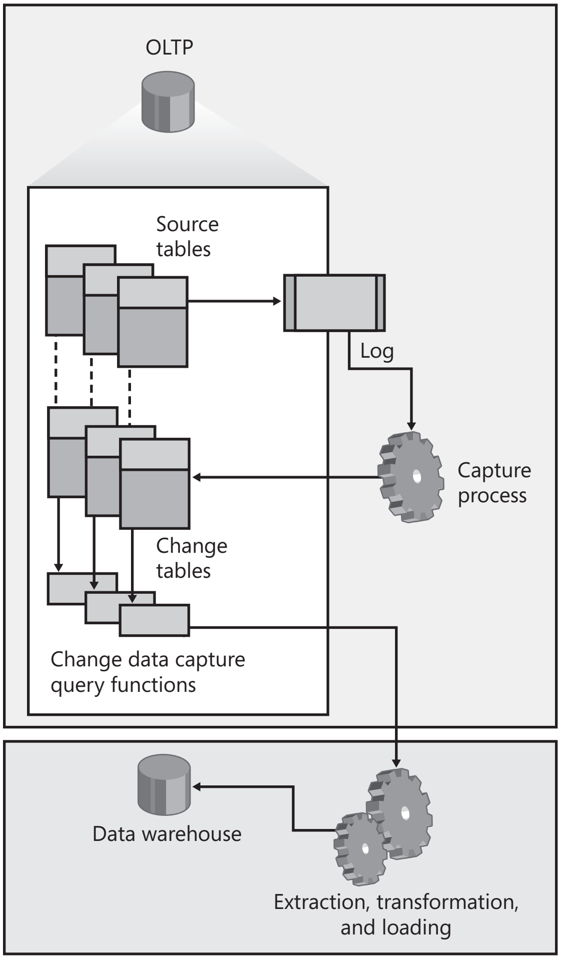

OLTP
Source tables
Log
Capture process
Change tables
Change data capture query functions
Data warehouse
Extraction, transformation, and loading

FIGURE 14-9 High-level architecture of CDC

Several new system stored procedures and table-valued functions (TVFs) are provided to enable, monitor, and consume SQL Server Change Tracking output. To begin, you execute the *sp_cdc_enable_db* procedure to enable CDC on the current database. (You must be in the *sysadmin* role to do this.) When you run this procedure, a new *cdc* user, *cdc* schema, and *CDC_admin* role are created. These names are hard-coded, so in the event that you already have a user or schema named *cdc*, you will need to rename it before using CDC.

Once the database is CDC-enabled, you enable CDC on a given table by executing *sp_cdc_enable_table*. (You must be in the *db_owner* role to do this.) When you do that, several objects are created in the *cdc* schema: a change table and at least one (but possibly two) TVFs. Let's look at each of these objects in turn.

When CDC is enabled on a table, SQL Server creates a change table in the *cdc* schema corresponding to the table on which CDC is being enabled. The change table will be populated with change data automatically by CDC and is assigned a name based on both the schema and the table being tracked. For example, when you enable CDC on the *Employee* table in the *dbo* schema (as we'll do shortly), SQL Server creates a corresponding change table named *cdc.dbo_Employee_CT* that will record all changes to the *dbo.Employee* table. The schema of the tracked table (*dbo* in this case) is part of the change table name so that same-named tables from different schemas can all be unambiguously tracked in the *cdc* schema. It is also possible to explicitly name the change table, as long as it's unique in the database.

The ETL process will query this change table for change data in order to populate your data warehouse, but it will not normally do so by selecting directly from it. Instead, the ETL process will call a special TVF to query the change table for you. This TVF is also created for you by SQL Server automatically when the change table is created, and—like the change table—the TVF is also created in the *cdc* schema with a name based on the schema and table name of the tracked table. So again, if we're tracking the *dbo.Employee* table, SQL Server creates a TVF named *cdc.fn_cdc_get_all_changes_dbo_Employee* that accepts parameters to select all changes that occur to *dbo.Employee* between any two desired points in time.

If you specify *@supports_net_changes=1* when calling *sp_cdc_enable_table*, a second TVF is created for the change table as well. Like the first TVF, this one allows you to select changes between any two points in time, except that this TVF returns just the *net* (final) changes that occurred during that time frame. This means, for example, that if a row was added and then deleted within the time frame being queried using this second TVF, data for that row would not be returned—whereas the first TVF would return data that reflects both the insert and the delete. This second TVF is named in a similar fashion as the first, except using the word *net* instead of *all*. For *dbo.Employee*, this TVF is named *cdc.fn_cdc_get_net_changes_dbo_Employee*. Note that querying for net changes requires the tracked table to have a primary key or unique index.

Neither of these TVFs accept start and end times directly but instead require the range to be expressed as log sequence numbers (LSNs) by first calling *sys.fn_cdc_map_time_to_lsn*. So to query between two points in time, you call *sys.fn_cdc_map_time_to_lsn* twice—once for the start time and once for the end time—and then use the LSN values returned by this function as input values to the TVFs for querying change data. This might seem unnecessarily cumbersome, but in fact has good reason related to supporting two change tables on the same table, one feeding the production systems and another supporting ongoing development.

Tip The start and end times this function is called with are not required to fall within the range of time actually represented in the log. If either time falls outside the boundaries in the log, the function "does the right thing": it returns the earliest existing LSN if the specified start time is prior to the earliest LSN, and it returns the latest existing LSN if the specified end time is after the latest LSN. This will be implicitly illustrated shortly in Listing 14-3 for both start and end times.

The *sp_cdc_enable_table* stored procedure has several optional parameters that give you a lot of flexibility. You can, among other options, specify your own name for the change table, a role that a user must belong to in order to query changes (if not in *sysadmin* or *db_owner*), which columns of the table should be tracked (you don't need to track all of them), the filegroup on which to create the change table, and whether the *SWITCH_PARTITION* option of *ALTER TABLE* can be executed against the tracked table (which has very important implications). Consult SQL Server Books Online for more details of *sp_cdc_enable_table* parameters.

When you no longer require CDC on a particular table, you can call the *sp_cdc_disable_table* stored procedure on the table. This procedure drops the change table and the TVFs and updates the system metadata to reflect that the table is no longer tracked. When you no longer require CDC on the database, call the *sp_cdc_disable_db* stored procedure to completely disable CDC for the entire database.

Important You should be aware of several considerations before dropping a database on which CDC has been enabled. To drop a CDC-enabled database, you must either stop SQL Server Agent or first disable CDC by running *sp_cdc_disable_db* on the database to be dropped. If you take the former approach, the SQL Server Agent jobs will be deleted automatically when SQL Server Agent is next started upon detecting that the database the jobs were associated with is no longer present. Of course, SQL Server Change Tracking for other databases running on the server instance will also be suspended while SQL Server Agent is stopped. The latter approach is the preferred method, since it does not interfere with other CDC-enabled databases and will remove all CDC artifacts related to the database being dropped.

The change table records all changes to the requested columns, including intermediate states (per DML statement) between two points in time. Note that CDC supports sparse columns (covered later in this section) but not sparse column sets. Each change table row also includes five metadata columns of great value for change-consuming processes to determine what type of change (insert, update, or delete) each row represents and to group and order all changes belonging to the same transaction. One item it c*annot* capture is who made the change, which is why it is not ideal for maintaining audit trails. For that, you can use SQL Server Audit, which will track and record which users are performing data modifications as well as any other activity of interest. (We cover SQL Server Audit in Chapter 5.)

As we mentioned earlier, CDC relies on SQL Server Agent for automating the capture process. The first time *sp_cdc_enable_table* is executed on any table in a database, SQL Server also creates two SQL Server Agent jobs for that database. The first is a change-capture job, which performs the actual transaction log monitoring to apply changes on the tracked table to the corresponding change table. The second is a cleanup job, which deletes rows from change tables after a configurable interval (three days, by default) and removes all CDC artifacts if the tracked table is dropped. Therefore, SQL Server Agent must be running the first time this procedure is run to CDC-enable a table on any database in the server instance. Subsequently, if SQL Server Agent stops running, changes to tracked tables will accumulate in the transaction log but not be applied to the change tables until SQL Server Agent is restarted.

CDC can at first appear rather cumbersome to use, but it is well thought out in terms of its configuration flexibility and support for various scenarios. Some of these might not be immediately obvious—for example, what happens if a tracked table is dropped, or its structure changed, after CDC is enabled on it? We lack the space to delve into these essential aspects, but you'll find comprehensive details in SQL Server Books Online. The code in Listing 14-3 shows a complete example of using CDC.

LISTING 14-3 Using Change Data Capture

```
-- Create test database
CREATE DATABASE CDCDemo
GO

USE CDCDemo
GO

-- Enable CDC on the database
EXEC sp_cdc_enable_db

-- Show CDC-enabled databases
SELECT name, is_cdc_enabled FROM sys.databases

-- View the new "cdc" user and schema
SELECT * FROM sys.schemas WHERE name = 'cdc'
SELECT * FROM sys.database_principals WHERE name = 'cdc'

-- Create Employee table
CREATE TABLE Employee(
 EmployeeId    int NOT NULL PRIMARY KEY,
 EmployeeName  varchar(100) NOT NULL,
 EmailAddress  varchar(200) NOT NULL)

-- Enable CDC on the table (SQL Server Agent *should* be running when you run this)
```

```
EXEC sp_cdc_enable_table
 @source_schema = N'dbo',
 @source_name = N'Employee',
 @role_name = N'CDC_admin',
 @capture_instance = N'dbo_Employee',
 @supports_net_changes = 1

-- Show CDC-enabled tables
SELECT name, is_tracked_by_cdc FROM sys.tables

-- Insert some employees...
INSERT INTO Employee VALUES(1, 'John Smith', 'john.smith@ourcorp.com')
INSERT INTO Employee VALUES(2, 'Dan Park', 'dan.park@ourcorp.com')
INSERT INTO Employee VALUES(3, 'Jay Hamlin', 'jay.hamlin@ourcorp.com')
INSERT INTO Employee VALUES(4, 'Jeff Hay', 'jeff.hay@ourcorp.com')

-- Select them from the table and the change capture table
SELECT * FROM Employee
SELECT * FROM cdc.dbo_employee_ct

-- Delete Jeff
DELETE Employee WHERE EmployeeId = 4

-- Results from Delete
SELECT * FROM Employee
SELECT * FROM cdc.dbo_employee_ct
-- (Note: result of DELETE may take several seconds to show up in CT table)

-- Update Dan and Jay
UPDATE Employee SET EmployeeName = 'Dan P. Park' WHERE EmployeeId = 2
UPDATE Employee SET EmployeeName = 'Jay K. Hamlin' WHERE EmployeeId = 3

-- Results from update
SELECT * FROM Employee
SELECT * FROM cdc.dbo_employee_ct      -- See note above

-- Give the CDC job a chance to initialize before accessing the TVFs
WAITFOR DELAY '00:00:20'

-- To access change data, use the CDC TVFs, not the change tables directly
DECLARE @begin_time datetime
DECLARE @end_time datetime
DECLARE @from_lsn binary(10)
DECLARE @to_lsn binary(10)
SET @begin_time = GETDATE() - 1
SET @end_time = GETDATE()

-- Map the time interval to a CDC LSN range
SELECT @from_lsn =
 sys.fn_cdc_map_time_to_lsn('smallest greater than or equal', @begin_time)

SELECT @to_lsn =
 sys.fn_cdc_map_time_to_lsn('largest less than or equal', @end_time)

SELECT @begin_time AS BeginTime, @end_time AS EndTime
SELECT @from_lsn AS FromLSN, @to_lsn AS ToLSN

-- Return the changes occurring within the query window.

-- First, all changes that occurred...
SELECT *
 FROM cdc.fn_cdc_get_all_changes_dbo_employee(@from_lsn, @to_lsn, N'all')

-- Then, net changes, that is, final state...
SELECT *
 FROM cdc.fn_cdc_get_net_changes_dbo_employee(@from_lsn, @to_lsn, N'all')
```

Let's examine this code closely. After creating our sample database *CDCDemo*, we enable CDC on that database by calling *EXEC sp_cdc_enable_db*. The next several *SELECT* queries demonstrate how to retrieve various kinds of CDC-related information. The first *SELECT* query shows how the *is_cdc_enabled* column in *sys.databases* returns true (*1*) or false (*0*), making it easy to find out which databases are CDC-enabled and which aren't. The next two *SELECT* queries show how the new *cdc* schema and user can be found in *sys.schemas* and *sys.database_principals*.

The code then proceeds to create the *Employee* table, which has only three columns to keep our example simple. CDC is then enabled on the *Employee* table by calling *EXEC sp_cdc_enable_table* and passing parameters that identify the *Employee* table in the *dbo* schema for change capture. (Remember that SQL Server Agent must be running at this point.) The next *SELECT* statement shows how to query the *is_tracked_by_cdc* column in *sys.tables* to find out which tables are CDC-enabled and which aren't.

Recall that enabling CDC on the *Employee* table creates a TVF for retrieving all changes made to the table between any two points in time. Recall too that by specifying *@supports_net_changes = 1*, this also creates a second TVF for retrieving only the *net* changes made between any two points in time. The difference between *all* changes and *net* changes will be very clear in a moment, when we call both of these TVFs and compare their results. But first the code performs a mix of *INSERT*, *UPDATE*, and *DELETE* operations against the *Employee* table to simulate database activity and engage the capture process. In Listing 14-3, these operations are accompanied by *SELECT* statements that query the change table *cdc.dbo_employee_ct*. This is done purely to demonstrate that change data for the *Employee* table is being captured to the change table. However, you should normally not query the change tables directly in this manner and should instead use the generated TVFs to extract change information about the *Employee* table, as demonstrated by the rest of the code.

Our code then executes a *WAITFOR* statement to pause for 20 seconds before calling the TVFs, in order to give the SQL Server Agent change capture job a chance to initialize. This is a one-time latency only; it does *not* represent the normal latency for CDC-tracked changes to be recorded, which is on the order of 2 to 3 seconds. Without this delay, or if SQL Server

Agent is not running when you call the TVFs, you will receive a rather misleading error message that unfortunately does not describe the actual problem.

To call either of the generated TVFs, you need to provide a value range that defines the window of time during which you want change data returned. As already explained, this range is expressed using LSN values, which you can obtain by calling *sys.fn_cdc_map_time_to_lsn* and passing in the desired start and end points in time. So first we establish a time range for the past 24 hours, which we obtain by assigning *GETDATE() – 1* and *GETDATE()* to the start and end time variables. Then we call *sys.fn_cdc_map_time_to_lsn* on the start and end time variables to obtain the LSN values corresponding to the last 24 hours. (Note that the starting LSN gets adjusted automatically to compensate for the fact that there are no LSNs from 24 hours ago, as does the ending LSN, since there might not be any from a moment ago either.) We then issue two *SELECT* statements so that we can view the time and LSN range values, an example of which is shown here:

```
BeginTime               EndTime
----------------------- -----------------------
2008-07-08 23:42:55.567 2008-07-09 23:42:55.567

(1 row(s) affected)

FromLSN                ToLSN
---------------------- ----------------------
0x0000001A0000001E0039 0x00000020000000A50001

(1 row(s) affected)
```

Equipped with the LSN range values, we issue two more *SELECT* statements. (These are the last two statements in Listing 14-3.) The first statement queries the range against the *all* changes TVF, and the second statement queries the range against the *net* changes TVF. Comparing the results of these two queries clearly illustrates the difference between the TVFs, as shown here:

```
__$start_lsn           __$seqval              __$operation __$update_mask EmployeeId
EmployeeName      EmailAddress
---------------------- ---------------------- ------------ -------------- ---------- -------
---------- -------------------------
0x0000001E0000007C0013 0x0000001E0000007C0012 2            0x07           1          John
Smith       john.smith@ourcorp.com
0x0000001E000000800003 0x0000001E000000800002 2            0x07           2          Dan
Park        dan.park@ourcorp.com
0x0000001E000000810003 0x0000001E000000810002 2            0x07           3          Jay
Hamlin      jay.hamlin@ourcorp.com
0x0000001E000000820003 0x0000001E000000820002 2            0x07           4          Jeff
Hay         jeff.hay@ourcorp.com
0x0000001E000000850004 0x0000001E000000850002 1            0x07           4          Jeff
Hay         jeff.hay@ourcorp.com
0x0000001E000001AC0004 0x0000001E000001AC0002 4            0x02           2          Dan P.
Park        dan.park@ourcorp.com
0x0000001E000001AE0004 0x0000001E000001AE0002 4            0x02           3          Jay K.
```

```
Hamlin      jay.hamlin@ourcorp.com

(7 row(s) affected)

__$start_lsn           __$operation __$update_mask EmployeeId EmployeeName      EmailAddress
---------------------- ------------ -------------- ---------- ----------------- ------------
--------------
0x0000001E0000007C0013 2            NULL           1          John Smith        john.smith@
ourcorp.com
0x0000001E000001AC0004 2            NULL           2          Dan P. Park       steven.
jones@ourcorp.com
0x0000001E000001AE0004 2            NULL           3          Jay K. Hamlin     jay.hamlin@
ourcorp.com

(3 row(s) affected)
```

The first result set includes all the information about all changes made during the specified LSN range, including all interim changes. Thus, the information returned from the first *all* changes TVF shows every stage of change, or seven changes in total. In our scenario, John was inserted once and then never changed. So only his insert (*__$operation* value *2*) is shown. Dan and Jay were inserted (*__$operation* value *2*) and updated (*__$operation* value *4*), so both changes (insert and update) are returned for each of them. Jeff, on the other hand, was deleted (*__$operation* value *1*) after being inserted, so both changes (insert and delete) are returned for Jeff.

The second result set includes only the *final* changes made during the specified LSN range. So for the same LSN range, we receive only three change records from the second *net* changes TVF, each of which provides the final column values in the specified LSN range. John appears only once as in the previous query, since he was inserted only once and never modified or deleted within the LSN range. However, although Dan and Jay were inserted and updated, they each appear only once (with their final values for the LSN range), and not twice as in the previous query. And since Jeff was inserted and deleted within the window of time specified by the LSN range, no change data for Jeff is returned at all by the *net* changes TVF.

Partitioned Table Parallelism

In SQL Server, a *partitioned table* is a table whose physical storage is divided horizontally (that is, as subsets of rows) into multiple filegroups (invisibly to queries and DML) for the purpose of improved manageability and isolation of various kinds of otherwise potentially conflicting access. For example, different partitions of the same table can have different backup and compression strategies and indexes, each optimized to the use of the partition. Given the large size of many data warehouses, this flexibility can be invaluable.

The typical (although by no means required) partition key is *Time*, since that is so often the natural horizontal dividing line. Partitioning by *Time* allows, for example, "old" data to be indexed more lightly than current, more frequently accessed data. Old data can also be backed up and deleted without affecting simultaneous queries against more recent data. Partitioning is an important tool of physical implementation, particularly when building a very large data warehouse.

Another potential benefit of well-designed partitioning is more efficient query plans. Queries specifying the partitioning key that involve only a single partition benefit from having less data to traverse (and potentially more optimized indexes if the partition is for newer data). In addition, when SQL Server is running on multiple-core or multiple-CPU hardware and configured appropriately, multiple worker threads are available and can achieve parallelism in processing a query by assigning multiple threads to it.

Note For maximum partitioning benefit, it is crucial to physically isolate each partition of a table from each of the others. In practice, this means that each filegroup of each partition should be on a different physical disk and, in extreme cases, even on a different disk controller. In general, however, this book does not explain the mechanics of partitioned tables, which are well covered in SQL Server Books Online.

Thread Management

SQL Server 2005 optimized parallelism for queries involving only a single partition, by allocating all available threads to the one partition. However, on a multipartition query, performance could suffer badly because then only one thread is allocated per partition—leading to some parallelism for the query as a whole but none per partition. The result was that queries varying only slightly in their partitioning key constraint could exhibit vastly different degrees of performance.

The new Partitioned Table Parallelism feature in SQL Server 2008 directly addresses this shortcoming by allocating all available threads to a multipartition query in round-robin fashion. The result is that each partition, as well as the query as a whole, achieves some degree of parallelism. This is automatic when applicable. The best gains will be achieved when the number of threads (that is, cores or CPUs) is significantly larger than the number of partitions on the table. The difference between SQL Server 2005 and 2008 in thread allocation for multipartition queries is illustrated in Figure 14-10. Under the latter in this example, three times as many threads per partition operate on the Feb YTD query, and with all else being equal, this should translate to a 200 percent performance improvement.

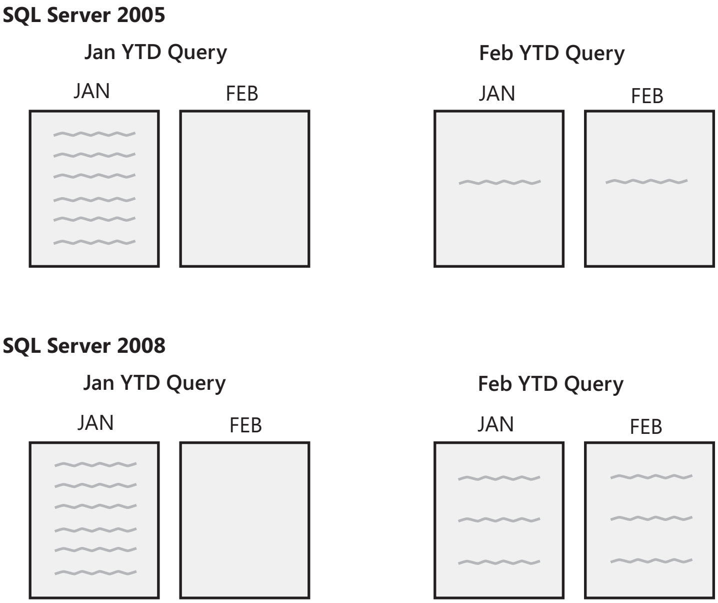


FIGURE 14-10 The difference between SQL Server 2005 and 2008 in how threads are allocated to multipartition queries

Note Partitioned Table Parallelism is available only in the Enterprise edition of SQL Server 2008.

Lock Escalation

Another important feature of Partitioned Table Parallelism relates to table locking behavior. Previously, when deciding whether to elevate to a table-level lock on a partitioned table, the database engine did not take into account whether concurrent statements against the same table were each accessing a different partition. When they were, each was logically independent and there would be no reason for one to block another. But by not recognizing this and escalating one of the statements to a table lock, the database engine could unnecessarily block the remaining statements, in the process also enhancing the possibility of deadlocks among them. In SQL Server 2008, the default behavior on a partitioned table behaves as before, but Partitioned Table Parallelism enables a new *ALTER TABLE* option, which directs the database engine to use partition-level lock escalation, instead of table-level, on a partitioned table. The syntax is shown here:

```
ALTER TABLE MyTable SET (LOCK_ESCALATION = <option>)
```

The *LOCK_ESCALATION* option can be specified as *TABLE*, *AUTO*, or *DISABLE*. The default is *TABLE*, which means that only table-level lock escalation will occur. If you specify *AUTO*, you get partition-level locking on partitioned tables, table-level otherwise. With *DISABLE*, no lock escalation will occur (in most cases).

Star-Join Query Optimization

Star-Join Query Optimization is an important new feature in SQL Server 2008 (again, available in Enterprise edition only) in the context of data warehouse–oriented performance enhancements, but it does not lend itself to deep explanation in a book like this because it does not offer any user-adjustable properties and its operation is largely buried within the database engine. The good news is that you need not do anything to get the benefit of it when applicable.

As noted earlier, the star schema is a common physical data model in Kimball-style data warehousing architectures. Queries against such a physical model are typically characterized by a central fact table joined to multiple dimension tables, each on single-column equijoins (joins based on equality), where the fact table has much higher cardinality than the dimension tables (more rows in the fact table as compared with the dimension table), and the constraints of the query are all on the dimension tables—a pattern now known as a *star-join*. Since this pattern is common across a large range of data warehousing scenarios, it became apparent that a query optimizer that could recognize such a pattern could potentially produce more efficient query plans than otherwise.

Here's the basic idea. Eliminate as many candidate rows from the fact table as early as possible in the query-resolution pipeline, since the fact table typically has by far the highest cardinality of the tables involved. In practice, this means determining the candidate join keys from the dimension tables first (taking advantage of the winnowing effect of the constraints typically on them) and then using this information to eliminate candidate rows from the fact table ahead of, and more efficiently than, the actual join process further down the pipeline would. The heuristics—or in other words the rules by which the optimizer recognizes a star-join query—are important to the effectiveness of this strategy.

Such mechanisms are complex and, for our purposes, largely opaque. SQL Server 2005 introduced some star-join optimization based on these principles, but SQL Server 2008 extends the degree to which it can recognize and optimize this pattern. Microsoft benchmarks assert that the degree of performance improvement on typical data warehouse queries at which this feature is targeted can range from 10% to 30%. The SQL Server 2008 enhancements in this area also include more detailed information in query plans, which help the designer to understand when or if this feature is being applied to particular queries.

Note This enhancement will be of most value when a significant part of the SQL Server workload involves ad hoc SQL queries against a star schema. If your architecture directs most ad hoc queries to an OLAP cube, it will be of lesser, if any, benefit, unless your cube is hosted by SQL Server Analysis Services and uses the Relational OLAP (ROLAP) or Hybrid OLAP (HOLAP) storage mode (since in these cases a significant number of cube queries might become SQL star schema queries).

Space considerations preclude us from discussing this feature in more detail here. To learn more, we recommend that you visit the links provided at the end of this section.

SPARSE Columns

Not all practitioners are happy with *NULL* values in a relational database schema, but for better or worse, they are widely used in practice. Without engaging that debate, some will rationalize allowing nullable columns when physically modeling a type (for example, *Product*) that has many subtypes that have few attributes in common and many attributes unique to each subtype. It can be convenient, despite going against the grain of normalization, to physically model this situation as a single table with a column for every attribute across all subtypes. In such a case, each attribute column must be nullable and will be sparse—that is, containing *NULL* in a high percentage of cases. It would be beneficial if the storage for such sparsely populated nullable columns were optimized, particularly in the data warehousing context, given the often large database sizes involved.

In versions earlier than SQL Server 2008, storing *NULL* values was not optimized—it required storage for every *NULL* occurrence. SQL Server 2008 introduces the notion of the *SPARSE* column, a nullable column whose storage is optimized for *NULL* values—at the cost of increased storage overhead for non-*NULL* values. With this option enabled, occurrences of *NULL* use no storage. (Note that this is also true when SQL Server Data Compression, detailed in the next section, is used—although the two are not equivalent.) The density of a column's *NULL* values required to achieve a 40 percent space saving using the *SPARSE* attribute, the nominal space saving value as reported by SQL Server Books Online, depends on the column's data type and ranges from 42 percent for 38-digit high-precision numeric types to 98 percent for *bit*. The *SPARSE* attribute in particular benefits Microsoft Office SharePoint Server, which by its generic and end-user-driven nature is a particular case of the preceding scenario—needing to store many user-defined attributes that are sparse by nature.

A few data types cannot be *SPARSE*, and there are other, potentially significant, restrictions on using *SPARSE* columns—for example, they cannot have default values or rules or be part of a clustered index or unique primary key index. SQL Server Books Online provides full details.

This feature is enabled by decorating column definitions in your *CREATE TABLE* and *ALTER TABLE* statements with the new *SPARSE* attribute. Obviously, the column must also be declared *NULL*. Listing 14-4 shows an example of usage.

LISTING 14-4 Declaring *SPARSE* columns

```
CREATE TABLE SparseTest
(ID       int IDENTITY(1,1),
 LastName varchar(50) SPARSE NULL,
 Salary   decimal(9,2) NULL)
GO

ALTER TABLE SparseTest
 ALTER COLUMN Salary decimal(9,2) SPARSE
GO
```

SQL Server 2008 introduces two other new features that have a relationship to the *SPARSE* feature but do not depend on it. The first is the *column set*, an optionally declared set of specified columns on a table that, once declared, associates an *xml* column with the table as metadata (that is, no additional storage is used). This column represents the specified columns as an XML document and allows querying *and* updating of the columns as a group using XQuery and XML DML (which we cover in depth in Chapter 6). The individual columns can still be referenced in the usual way, but the column set representation can be a more convenient method when a table has a large number of columns and might provide performance improvements in some cases. *SPARSE* columns relate to column sets in that a column set cannot be added to an existing table already containing any *SPARSE* columns, and if *SPARSE* columns are later added to a table with a column set, they automatically become part of the column set.

The second new feature is the *filtered index*. A filtered index is an optimized nonclustered index whose declaration includes a *WHERE* clause that restricts the values included in the index to those specified. This can have wide-ranging implications for index maintenance, index storage, and query plan optimization. This feature is most useful when the query patterns against the table are well understood and they naturally relate to distinct subsets of rows. *SPARSE* columns are good candidates to participate in a filtered index because they represent distinct, well-defined subsets (rows with *NULL*s in the columns and rows with non-*NULL*s). For more details of both these features, which involve considerable complexity in their own right, see SQL Server Books Online.

A final benefit of *SPARSE* columns is that, by their nature, they can reduce the size of large backups, potentially more so than any of the new compression features we cover in the next section.

Data Compression and Backup Compression

Data compression and backup compression are long-awaited enhancements to SQL Server—not surprisingly, also available only in the Enterprise edition (with one exception, as we'll see when we discuss backup compression). They are of benefit in all scenarios, but especially for large data warehouses. Many factors cause a data warehouse to grow at least linearly with time: the desire to facilitate trend analyses, personalization, and data mining; the fact that most data warehouses increase the number of data sources included over time; and last that multiple copies of the data warehouse often exist for redundancy and development and QA purposes. SQL Server 2008 provides both data compression, targeting the database itself, and backup compression, targeting the backup/restore process.

As the size of the data warehouse increases, it affects the cost and complexity of maintaining the online version and of taking backups of it. SQL Server 2008 Data Compression provides many benefits. It aids online query performance by increasing the number of rows stored per page, lessening disk I/O and saving costs in disk space. It improves performance for a given amount of memory, as more rows can be held in memory at the same time. It can benefit the backup/restore process by minimizing the I/O and therefore time and media required, since less physical data needs to be transferred. Last, replication and mirroring scenarios can also benefit for all the same reasons.

Data Compression

SQL Server 2005 made a start at targeting data compression concerns with both its table-level *vardecimal* storage format (in Service Pack 2 for the Enterprise edition) and its ability to use NTFS file system file compression on SQL Server read-only secondary data files (or all files, including log files, if the database is read-only).

These enhancements remain supported in SQL Server 2008, although use of the *vardecimal* option is deprecated and use of NTFS compression for SQL Server data files is mostly not recommended. Instead, SQL Server 2008 goes considerably beyond these earlier enhancements in the features it provides for data compression.

The most basic form of data compression uses a storage format that eliminates unneeded precision in fixed-length data types—that is, representing each value in a column with the minimal number of bits necessary. For example, any value of 255 or less stored in an integer data type could be stored in one byte instead of four (neglecting some slight overhead). SQL Server 2005 provided such compression or variable-length storage only for the *decimal* and *numeric* data types, but SQL Server 2008 provides it for *all* formerly fixed-length data types (including *decimal* and *numeric*). Note that what is changing is storage format, not data type, so the semantics of each data type remain the same to T-SQL queries as well as applications.

Data compression comes in two forms: *row compression* (RC) and *page compression* (PC). RC is another name for the variable-length storage approach just detailed. With RC, all occurrences of *0* (zero) and *NULL* consume no space. RC is not effective for variable-length data types (they are already effectively compressed), for some shorter data types (where the overhead of compression outweighs the benefit), and for some other data types for technical reasons.

Note To summarize, RC does *not* apply to *tinyint*, *smalldatetime*, *date*, *time*, *varchar*, *text*, *nvarchar*, *ntext*, *varbinary*, *image*, *cursor*, *sql_variant*, *uniqueidentifier*, *table*, *xml*, and user-defined types (UDTs).

PC is a superset of RC and provides potentially greater overall compression than RC alone, at the cost of greater CPU overhead. Where RC is concerned with compressing scale and precision on each individual row-column value, PC is concerned with compressing redundancy across all the rows and their columns on a particular page. PC can be used with all the same database objects as RC. It applies three steps to the enabled object, in the order indicated:

1. RC to the leaf level of a table and to all levels of an index.
2. PC—on each page, for each column of each row on that the page, any common prefixes among all values stored in that column (if any) are identified and tokenized. Each such prefix value is stored once in the new Compression Information (CI) section of the page (by column), and values in each column are replaced with short encoded values that identify the prefix and how much of it applies (as a prefix to the remainder of the value).
3. Dictionary compression—on each page, repeating values from any column in any row on the page are identified and stored in the CI area, and the values are replaced with a pointer to the repeated value. This can further compress the results of the first two steps.

As data is added to a PC-enabled object, these operations are initiated only when a page becomes full. If PC is enabled on an existing object containing data, that object must be rebuilt, a potentially expensive operation.

The code in Listing 14-5 shows an example of creating a table enabled for PC.

LISTING 14-5 Enabling PC on a table

```
CREATE TABLE RowCompressionDemo
 (FirstName char(10),
  LastName  char(30),
  Salary    decimal(8,2))
 WITH (DATA_COMPRESSION = PAGE)
```

SQL Server 2008 provides a system stored procedure associated with both forms of compression aptly named *sp_estimate_data_compression_savings*, which can be used to evaluate whether compression is worth applying to a given object. It can be run for a given uncompressed table, index, or partition to estimate the size it would be, using both RC and PC. It can also do the reverse; reporting the size a compressed object would be if uncompressed. This procedure works by sampling the data of the indicated object into a temporary store and running the indicated compression or decompression on it. It is possible for it to report a larger size for compressed than uncompressed data, which indicates clearly that the nature of the data is such that the storage overhead associated with compression outweighs any benefit.

Of course, these forms of compression require more CPU cycles to use than would otherwise be required, both when writing (compressing) and reading (decompressing) data. Each represents a tradeoff between saving space (disk and memory) and increasing CPU use. In addition, the effectiveness of any compression scheme is sensitive to the data type and statistical distribution of the values being compressed. For example, compression of an *int* column (4 bytes) in which most values do not exceed 255 (which fit in 1 byte) would exhibit much more benefit from RC than if the values were evenly distributed or if the column were already declared as a *tinyint* (1 byte). For these reasons, as well as the fine grain of data types that this feature allows to be individually tuned for compression, it is advisable to experiment with the various compression options to determine the optimal combination of settings.

Data compression must be enabled—it is disabled by default. It can be enabled on an entire table (which applies to all of its partitions), on individual partitions of a table, on individual indexes of a table, on individual index partitions, and on the clustered index of an indexed view. These features, together with the separately selectable options of row or page compression, give the database administrator great flexibility in tuning the use of compression to achieve the best tradeoffs.

Data compression is enabled by *CREATE TABLE* (as shown earlier) and *CREATE INDEX* statements, and also by *ALTER TABLE* and *ALTER INDEX*. Note that SQL Server Data Compression is *not* automatically enabled on existing or subsequently created nonclustered indexes of a table on which data compression is enabled—each such index must be separately and explicitly enabled. The one exception to this is that a clustered index does inherit the compression setting of its table.

Last but not least, an uncompressed table can be rebuilt with either form of compression via the new *ALTER TABLE…REBUILD WITH (DATA_COMPRESSION=xxx)* statement, where *xxx* is either *ROW* or *PAGE*. As the compression process is CPU intensive, it lends itself to parallelism, and SQL Server 2008 can take advantage of the availability of multiple CPUs. The *REBUILD* clause therefore supports a *MAXDOP* option to control how many CPUs are allocated to the process.

Backup Compression

SQL Server Backup Compression is a new option with the *BACKUP* statement. Although only the Enterprise edition can create a compressed backup, any edition can restore one.

Compared with data compression, backup compression is extremely coarse grained. It is either enabled or it isn't for the entire backup—there are no options to tune the compression, and the compression methods are opaque. Nevertheless, it is a welcome enhancement since no earlier version of SQL Server provided any form of backup compression, forcing practitioners to compress backups in a separate step with other, non–SQL Server, utilities.

The option is disabled by default, but the default can be changed via server-level configuration or overridden in the *BACKUP* statement. It should be noted that an *uncompressed* 2008 backup operation (both create and restore) can benefit when SQL Server Data Compression has been used on a significant scale in the database being backed up, as a direct result of reduced I/O. If data compression *has* been used, backup compression will likely provide a smaller (possibly much smaller) space-saving benefit, and because of the additional CPU overhead, backup/restore time might perform worse than without backup compression. This feature is therefore most valuable when the database being backed up has not had significant data compression applied—your own experimentation is warranted.

> **Note** Compressed and uncompressed backups cannot be mixed in a backup media set.

As a simple example of the potential efficiency of backup compression, compare the size and time required to back up and restore the *AdventureWorksDW2008* database, as shown in Listing 14-6. The *CHECKPOINT* and *DBCC DROPCLEANBUFFERS* statements are used to ensure that all cache buffers are empty so that one test does not misleadingly improve the performance of the next. Create the directory C:\Backups prior to running the following code.

LISTING 14-6 Comparing the time and size between compressed and uncompressed backups

```
CHECKPOINT
DBCC DROPCLEANBUFFERS
BACKUP DATABASE AdventureWorksDW2008 TO DISK='C:\Backups\AWDWUncompressed.bak'
-- 10.661 sec, 71 Mb

CHECKPOINT
DBCC DROPCLEANBUFFERS
BACKUP DATABASE AdventureWorksDW2008 TO DISK='C:\Backups\AWDWCompressed.bak'
 WITH COMPRESSION
-- 6.408 sec, 13 Mb

CHECKPOINT
DBCC DROPCLEANBUFFERS
RESTORE DATABASE AWDWUncompressed FROM DISK = 'C:\Backups\AWDWUncompressed.bak'
 WITH MOVE 'AdventureWorksDW2008_Data' TO 'C:\Backups\AWDWUncompressed.mdf',
      MOVE 'AdventureWorksDW2008_Log' TO 'C:\Backups\AWDWUncompressed.ldf'
-- 9.363 sec

CHECKPOINT
DBCC DROPCLEANBUFFERS
RESTORE DATABASE AWDWCompressed FROM DISK = 'C:\Backups\AWDWCompressed.bak'
 WITH MOVE 'AdventureWorksDW2008_Data' TO 'C:\Backups\AWDWCompressed.mdf',
      MOVE 'AdventureWorksDW2008_Log' TO 'C:\Backups\AWDWCompressed.ldf';
-- 6.101 sec
```

In this example, you can see that there is much more improvement in the backup (compression) stage than the restore stage, but in both cases, performance for the compressed backup is superior to the uncompressed backup. This is due to the reduction in I/O required for processing the smaller (compressed) backup file. Of course, experiments are warranted in your particular scenario to determine exactly what improvements you will see for yourself.

Learning More

We've made several references to SQL Server Books Online for more detailed information about many of the new data warehousing features in SQL Server 2008. In addition, you can learn more about all of these SQL Server 2008 data warehousing–oriented features by visiting the following links:

- *http://msdn.microsoft.com/en-us/library/cc278097(SQL.100).aspx#_Toc185095880*
- *http://technet.microsoft.com/en-us/magazine/cc434693(TechNet.10).aspx*

These links were valid as of press time, but if they don't work, you can perform a Web search on "SQL Server 2008 data warehouse enhancements."

We can also recommend these additional resources to learn more about the recommended practices of data warehousing:

- *Building the Data Warehouse*, 4th ed., W. H. Inmon (Wiley, 2005)
- *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*, 2nd ed., Ralph Kimball and Margy Ross (Wiley, 2002), and *The Data Warehouse Lifecycle Toolkit*, Ralph Kimball et al. (Wiley, 2008)
- The Data Warehousing Institute, *http://www.tdwi.org/TDWI*

Summary

Data warehousing has become a key component of any enterprise-wide data architecture and is no longer only practical for the largest enterprises. Data warehousing developed as a response to the many impediments to creating actionable information from the data collected by operational applications, impediments that only gradually became recognized as significantly undermining the potential of computers to help turn data into information. The issues existed not only because of historical technical limitations but also because of fundamental differences in optimum design between operational and informational applications.

A data warehouse provides the foundation for many data-driven applications. SQL Server 2008 provides a full-featured, powerful, and cost-effective platform on which to build a data warehouse. You've seen how SQL Server 2008 is particularly targeted to data warehousing issues and provides a number of long-awaited features in that sphere. In addition, Microsoft also offers a wide range of integrated and complementary technology, including Microsoft Office SharePoint, Microsoft Performance Point, and the 2007 Microsoft Office system, which enable you to build informational applications on top of your SQL Server data warehouse foundation.